UNIVERSITY OF CALIFORNIA
Santa Barbara

Multi-Version Search and Cache-Conscious Ranking Optimization

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xin Jin

Committee in Charge:

Professor Tao Yang, Chair

Professor Jianwen Su

Professor Xifeng Yan

September 2017

The Dissertation of
Xin Jin is approved:

Professor Jianwen Su

Professor Xifeng Yan

Professor Tao Yang, Committee Chairperson

March 2017

Multi-Version Search and Cache-Conscious Ranking Optimization

by

Xin Jin

To my grandparents, parents and my wife.

Acknowledgements

First of all, I would like to express my most sincere thanks to my Ph.D. advisor, Prof. Tao Yang, for his continuous guidence, support and endless help during my Ph.D. study. I am amazingly lucky to have an advisor who is so great! He is always there to provide insights and help both in research and in my daily life. In the past five years, Tao has taught me how to conduct good research with his patience, enthusiasm and immense knowledge. Specifically, he gives me so much freedom in pursuing my personal interest in research fields, which makes me enjoying my projects so much. Also, Prof. Tao Yang has lots of fruitful experience in both academia and industry. Guided by him as my advisor, I believe the things I learned during my Ph.D. study will benefit a lot for my entire life. His continous guidence, care and support is so essential to help me complete my Ph.D.

Secondly, I want to give my sincere thanks to my Ph.D. committee members Prof. Jianwen Su and Prof. Xifeng Yan for their continuous guidence throughout my Ph.D. Jianwen and Xifeng have been my committee members for the MAE (Major Area Exam), proposal and defense. They have given me a lot of valuable feedback and insightful suggestions on my work. I am deeply grateful to them.

I benefitted a lot from the three internships at different companies. I worked with Dr. Wei Wang and Dr. Gang Zhao at Electronic Arts on a user-player relationship system optimization project, collaborated with Dr. Yun Xiao and Dr. Xingjian Zhang at Yahoo! search team on a second-phase ranking project, and worked with Dr. Srinath Rao and Kelvin Lim at Google Express search team on a merchant query redirection project and a search results deduplication project. I feel so lucky to have a chance to work with so many talented engineers and researchers and l really learned a lot from them.

My sincere thanks also go to all members in our great laboratory. I was having a wonderful time with you all, and greatly appreciate the collaboration with Xun Tang, Michael Daniel Agun, Qinghao Wu, Yifan Shen, Susen Zhao, Jiyu Chen, Nimisha Srinivasa and Lin Chai. Thanks for our discussion and debates on the projects, where a lot of new ideas coming up and problems solved. I am also grateful to other previous and current labmates and my friends at UCSB: Wei Wang, Maha Alabduljalil, Jinjin Shao, Kun Wan, Olaoluwa Osuntokun, Fangqiu Han, Yang Li, Huan Sun, Nan Li, Shengqi Yang, Shenghao Li and Jing Hao. Thanks for your encouragement and sharing along the way, which fullfills my memories during my Ph.D. study.

I also want to express my great thanks to all my colloborators. It is so good to be able to work with all these outstanding researchers in the world. I would like to thank Prof. Stefano Tessaro for your great suggestion and discussion, Dr. Claudio

Lucchese for your comments and data/codebase sharing, and also colloborators at my undergraduate study: Prof. Bin Cui, Dr. Junjie Yao and Yuxin Huang.

I own my deepest thanks to my families and closest friends. Thanks for my dear grandparents Qingxian Ma and Tianen Chen, you have taught me how to be a great person and how to love. Thanks for my dear parents Yi Chen and Jimin Jin for your endless support, encouragement and care. Thanks for your unconditional love in my life. And Qingyun, my dear wife, you are always there listening to me, caring me and sharing happiness and sadness with me. It is so great to have your support by my side all the time. Finally, I want to thank my closest friends. Thanks for accompanying me along the journey and provide helps whenever it is and wherever we are. Thank you all, for letting me feel that I am the luckist person in the world.

This dissertation study was supported in part by NSF IIS-1528041 and IIS-1118106. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

To all of them I dedicate this dissertation.

Curriculum Vitæ

Xin Jin

Education

2011-2017	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2010-2011	One year of graduate study at School of Computing, National University of Singapore, Singapore.
2006-2010	B.S. in Computer Science, Peking University, Beijing.

Field of Study

Major Field	Computer Science with Professor Tao Yang.

Professional Experience

09/2011-03/2017	Research Assistant, University of California, Santa Barbara.
06/2016-09/2016	Software Intern, Google, Mountain View, CA.
06/2015-09/2015	Software Intern, Yahoo!, Sunnyvale, CA.
06/2014-09/2014	Software Intern, Electronic Arts, Redwood City, CA.

Publications

Xin Jin, Tao Yang, and Xun Tang, Comparison of Cache Blocking Methods for Fast Execution of Ensemble-based Score Computation, *in Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval* (**SIGIR 2016**).

Xin Jin, Daniel Agun, Tao Yang, Qinghao Wu, Yifan Shen, and Susen Zhao, Hybrid Indexing for Versioned Document Search with Cluster-based Retrieval, *in Proceedings of the 25th ACM International Conference on Information and Knowledge Management* (**CIKM 2016**).

Xun Tang, Maha Alabduljalil, **Xin Jin**, and Tao Yang, Partitioned Similarity Search with Cache-Conscious Data Traversal, *in journal of ACM Transactions on Knowledge Discovery from Data* (**TKDD 2015**).

Xun Tang, **Xin Jin (equal contribution with the first author)**, and Tao Yang, Cache-Conscious Runtime Optimization for Ranking Ensembles, *in Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval* (**SIGIR 2014**).

Maha Alabduljalil, Xun Tang, **Xin Jin**, and Tao Yang, Load Balancing for Partitioned-based Similarity Search, *in Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval* (**SIGIR 2014**).

Junjie Yao, Bin Cui, Yuxin Huang, and **Xin Jin**, Temporal and Social Context based Burst Detection from Folksonomies, *in Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (**AAAI 2010**).

Daniel Agun, **Xin Jin**, Jinjin Shao, and Tao Yang, Efficient Privacy-Preservation Top K Search with Multi-Keyword Ranking, to be submitted for publication.

Xin Jin, Lin Chai, and Tao Yang, Cache-Conscious Ranking Optimization with Fast Parameter Selection for Ensemble-based Score Computation, to be submitted for publication.

Abstract

Multi-Version Search and Cache-Conscious Ranking Optimization

Xin Jin

Organizations and companies archive many versions of digital data such as web pages, internal emails and so on. Such data is critical for internal investigation, regulatory compliance, and electronic discovery. It is estimated that electronic discovery market that leverages archival data will reach $9.9 billions globally in 2017. It is not uncommon for many businesses to retain archived collections for 10 to 15 years. How to archive these versioned data is worth to study and we are facing many challenges including 1) traditional index occupies too much space for versioned data, 2) traditional search is too slow on versioned data, and 3) how to guarantee high accuracy when improving efficiency in new architecture.

In this dissertation, we take the opportunity of the fast developement of information retrieval and tackle the problem by proposing a new multi-version search architecture with cache-conscious ranking optimization framework. Specifically, we will first discuss our new versioned search architecture. Then, we will talk about a cache-conscious online ranking algorithm to improve the online part. Finally, we will describe a framework to select best blocking methods and parameters for our algorithm to achieve best performance.

Firstly, we present our new multi-version search architecture. We propose an approach that uses cluster-based retrieval to quickly narrow the search scope guided by version representatives at Phase 1 and develops a hybrid index structure with adaptive runtime data traversal to speed up Phase 2 search. The hybrid scheme exploits the advantages of forward index and inverted index based on the term characteristics to minimize the time in extracting positional and other feature information during runtime search. We compare several indexing and data traversal options with different time and space tradeoffs and describe evaluation results to demonstrate their effectiveness. The experiment results show that the proposed scheme can be up-to about 4x as fast as the previous work on solid state drives while retaining good relevance.

Secondly, we talk about our 2D blocking algorithm to optimize the online ranking part of the system. Multi-tree ensemble models have been proven to be effective for document ranking. Using a large number of trees can improve accuracy, but it takes time to calculate ranking scores of matched documents. We investigate data traversal methods for fast score calculation with a large ensemble and propose a 2D blocking scheme for better cache utilization with simpler code structure compared to previous work. The experiments with several benchmarks show significant acceleration in score calculation without loss of ranking accuracy.

Lastly, we describe a framework to fast select best blocking methods and parameters for our 2D blocking algorithm with the help of a full cache analysis. 2D blocking method is very helpful to improve online search efficiency. However, different traversal methods and blocking parameter settings can exhibit different cache and cost behavior depending on data and architectural characteristics. It is very time-consuming to conduct exhaustive search for performance comparison and optimum selection. We provide an analytic comparison of cache blocking methods on their data access performance for an approximation and propose a fast guided sampling scheme to select a traversal method and blocking parameters for effective use of memory hierarchy. The evaluation studies with three datasets show that within a reasonable amount of time, the proposed scheme can identify a highly competitive solution that significantly accelerates score calculation.

In summary, we have proposed a new multi-version search architecture with cache-conscious ranking optimization for the online search part and a framework to help fast select best blocking methods and parameters with full cache analysis for the 2D blocking method. By proposing this new versioned search system, we can meet challenges from scalability, efficiency and accuracy in multi-version search, and we believe this work would be useful to future researchers in this direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Organizations and companies archive many versions of digital data such as web pages, internal emails, source code, test data, and multimedia documents. Such data is critical for internal investigation, regulatory compliance, and electronic discovery [40]. For example, it is estimated that electronic discovery market that leverages archival data will reach $9.9 billions globally in 2017 [90].

It is not uncommon for many businesses to retain archived collections for 10 to 15 years and in some industries the retention periods may be as long as 100 years or more [42]. A long period of data retention implies many versions may have to be maintained. The size growth of an archival dataset also becomes rapid with advancements in cloud computing, content authoring, media sharing, and low-cost computer devices. Organizations also frequently backup their data. One

of the largest archival databases is Internet Archive [64] which has collected and preserved more than 376 billion web pages in the last decade with tens of petabytes of data.

Can backup files be used as an archive - all the information is in there, isn't it? It is; however there is a big difference between backup and archiving [63]. Backups are designed for recovery while archives require a sophisticated support for information retention, discovery and search. Discovery requirements grow every year, and backups simply cannot meet them. Archives are critical for some unanticipated legal or regulatory event and help discover details of information.

Recently several companies have been developing an integrated backup and archiving solution. For example, EMC data domain [41], NetApp [82], HP [61] and IBM SmartCloud Archive [94] are developed to consolidate backup, archive, and disaster recovery with high-speed, inline deduplication. However, still efficient support for complex search and analysis on archival data is not extensively studied and there are limited progress in this field.

To meet the above challenge, we present a representative cluster-based two-phase framework as the new versioned search architecture. Also, among the different pieces of a search architecture, one of the most important components that we take into attention is the online ranking part. Ranking is the key component

because it is the core of online service and online service is the one that faces users directly.

Ensemble-based machine learning techniques have been proven to be effective for dealing data-intensive applications with complex features and document ranking is a representative application benefiting from use of the large number of ensembles. For example, in the Yahoo! learning-to-rank challenge [33], all winners have used some forms of gradient boosted regression trees, e.g. [48]. The total number of trees reported for ranking can be upto 3,000 to 20,000 [52, 24, 53], or even 300,000 or more using bagging method [87]. Ranking for large ensembles is expensive. As reported in [99], it takes more than 6 seconds to rank the top-2000 results for a query processing a 8,051-tree ensemble and 519 features per document on an AMD 3.1 GHz core. If such an algorithm is used to compute scores for a large number of vectors in applications such as classification, the total job is also very time consuming. It takes even more time for a larger ensemble or for more candidate documents. The ranking process can be parallelized and the time can be reduced. However, it does not help for improving query throughput because less queries are processed in parallel. To tackle this problem, we propose a cache-conscious ranking framework to improve query throughput without affecting ranking accuracy.

1.1 Dissertation Overview

My research work aims to address the above challenges in multi-version search and cache-conscious ranking optimization, and the statement of the dissertation is as follows:

By proposing a versioned data search framework combined with cache-conscious ensemble ranking, we can build new systems to meet challenges from scalability, efficiency and accuracy.

Driven by the statement, the following are the three goals of this dissertation. Firstly, we take an overall look at the whole multi-version search problem, and propose a solution to versioned search using a hybrid indexing representative-guided two phase architecture. Then, we look into the ranking component of the online part of the search architecture by proposing a cache-conscious 2D method, which can improve ranking speed significantly comparing the state-of-the-art baseline method. Finally, since parameter setting in our 2D algorithm is huge important but it takes too much time using a brute force method, to guide our parameter setting, we propose a cache-analysis framework to help sample and select the best parameters efficiently. In the following, we briefly introduce the work included in this dissertation.

1.2 Multi-version Search System

The previous work on versioned data has studied the compression algorithms to identify shared data fragments among different versions [118]. Even though the index space for a versioned data collection can be compressed dramatically through fragmentation and deduplication, it is still time consuming to search the full index structure because a search procedure still has to deal with a large number of documents with many versions. A two-phase approach [56, 89] has been proposed to find top results first using a non-positional index and then rerank the selected top results with a positional index. Still there is a large number of versions to go through in Phase 1 even without a positional index.

This work is focused on processing conjunctive keyword queries on versioned datasets and our key idea is to extend the concept of cluster-based retrieval [4, 73, 75, 106] for representative-guided two-phase search and develop a per-cluster hybrid index to localize data access at Phase 2. Using representatives of document versions with full positional information reduces the number of top clusters needed to retain a good relevancy. The tradeoff is that Phase 2 requires memory caching of index or the use of solid state drives (SSD). To speedup Phase 2 search, we develop hybrid per-cluster indexing with adaptive traversal of forward and inverted structure. Our evaluation shows that the proposed scheme is up-to

about 4x as fast as the two-phase approach [56] when the search index is available from memory or from an SSD.

1.3 Cache-Conscious Runtime Optimization for Ranking Ensembles

Computing scores from a large number of trees is time-consuming. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and instruction branch prediction. Compiler optimization [13] cannot handle complex code such as rank scoring very well. For example, processing a 8,051-tree ensemble can take up to 3.04 milliseconds for a document with 519 features on an AMD 3.1 GHz core. Thus the scoring time per query exceeds 6 seconds to rank the top-2,000 results. It takes more time proportionally to score more documents with larger trees or more trees and this is too slow for interactive query performance. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel.

Tradeoff between ranking accuracy and performance can be played by using earlier exit based on document-ordered traversal (DOT) or scorer-ordered traversal (SOT) [26], and by tree trimming [11]. The work in [12] proposes an

architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch mis-prediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient.

Unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. How can fast multi-tree ensemble ranking with simple code structure be accomplished via memory hierarchy optimization, without compromising ranking accuracy? This is the focus of this work.

We propose a cache-conscious 2D blocking method to optimize data traversal for better temporal cache locality. Our experiments show that 2D blocking can be up to 620% faster than DOT, up to 245% faster than SOT, and up to 50% faster than VPred. After applying 2D blocking on top of VPred which shows advantage in reducing branch mis-prediction, the combined solution Block-VPred could be up to 100% faster than VPred. The proposed techniques are complementary to previous work and can be integrated with the tree trimming and early-exit approximation methods.

1.4 A Comparison of Cache Blocking Methods

The previous work addressed the speedup of runtime execution for ensemble-based ranking in several aspects including tree trimming [11] for a tradeoff of ranking accuracy and performance, earlier exit [27], and loop unrolling [12], and ensemble restructuring for a tree-based model [76]. Memory access can be 100x slower than L1 cache and un-orchestrated slow memory access incurs significant cost, dominating the entire computation. The work shown in [99, 76] proposes a cache-conscious blocking method for better cache locality. However, there are other block methods to select and it is an open problem how to identify the best cache blocking method and parameter settings given different data and architecture characteristics.

Experimentally determining this choice can be extremely time-consuming and the comparative result may not be valid any more with a change of underlying feature vector structure or architecture. This work provides an analysis of multiple blocking methods with different data traversal orders, which provides better insight on program execution performance and leads a fast approximation to select the optimized structure.

Here, we consider the fast computation of ensemble-based scoring that aggregates and derives final scores for n feature vectors using m ensembles. For testing and comparing performance in ranking q sampled queries, the time cost

for searching through all combinations can be as high as $O(m^2 * n^2 * q)$. The main contribution of this work is to develop an analytic framework to compare memory access performance of data traversal under multi-level caches to find the fastest program execution with effective use of memory hierarchy. Our scheme results in a much smaller complexity with $O(m * n * q)$. Our experiments with three datasets corroborate the effectiveness of search cost reduction while the guided approximation identifies a highly competitive blocking choice. We also demonstrate the use of this scheme with QuickScorer [76] and for batched query processing.

1.5 Contributions

In this dissertation, there are three key contributions to the study of multiversion search with cache-conscious ranking optimization.

- Firstly, we design and propose a hybrid indexing representative-guided two phase architecture. In particular, the main constribution is a hybrid indexing method with adaptive runtime traversal in supporting fast two-phase versioned data search and an integration with cluster-based retrieval using guided representatives. Our evaluation with a prototype implementating using three datasets shows that we can be up-to 4.12x as fast as the baseline method.

- Secondly, we propose a cache-conscious design for computing ranking scores with a large number of trees and/or documents by exploiting memory hierarchy capacity for better temporal locality. Multi-tree score calculation of each query can be conducted in parallel on multiple cores to further reduce latency. The 2D method proprosed increase efficency tremendously but do not affect accuracy. Our experiment results show that we can be 50% faster than VPred, which is the state-of-the-art method in this field.

- Lastly, we develop a fast comparison and selection scheme to find an optimized cache blocking method for the 2D method we proposed with guided sampling. Our analysis estimates the data access cost of different methods approximately, which provides a foundation to select sampling points in comparing different methods and in narrowing search space. By using this selection framework, we can reduce the parameter finding time from thousand of years using brute force or tens of days using naive sampling to several hours.

1.6 Thesis Organization

The rest of the thesis is organized as follow.

In Chapter 2, we describe the design of the multi-version search system. We begin with the background of fragment-based index and introduce the necessity of using it. Then, we talk about our design considerations and present the cluster-based two-phase versioned search framework. After that, we elaborate our algorithm on choosing representatives, and we talk about our hybrid per-cluster indexing and traversal methods in detail. Finally, we give evaluations results on three datasets to show the advantage of our two-phase search architecture.

In Chapter 3, we present the 2D cache-conscious runtime optimization algorithm for ranking ensembles. We begin with the state-of-the-art famous learning-to-rank algorithm Gradient Boosted Regression Tree, and point out the possiblity to increase ranking efficiency by exploiting data/model cache locality. Then, we introduce different traversal patterns and talk about their data access difference. After that, we present our 2D block algorithm which can achieve much better cache locality by partitioning data and model into small blocks. Finally, we give our evaluation results to show the significant speedup of the proposed algorithm.

In Chapter 4, we elaborate our efforts in introducing a comparative analysis framework on different blocking methods. We begin with the design considerations and introduction of different blocking methods: DSD, SDS, SDSD and DSDS. Next, we give thorough cost analysis on each method under different scenarios. Then,we do a cost comparison and present our algorithm in choosing best method

and parameters. Finally, we show in the experiments that our analysis is indeed helpful in choosing parameters efficiently. We also provide evidence that our framework is a generatic one: it not only fits our 2D block algorithm, but also can be used on other algorithms like QuickScorer.

At the end of the dissertation, we conclude the work, and discuss future directions.

Chapter 2

Multi-version Search System

2.1 Introduction

Organizations and companies archive many versions of digital data such as web pages, internal emails, source code, test data, and multimedia documents. Such data is critical for internal investigation, regulatory compliance, and electronic discovery [40]. For example, it is estimated that electronic discovery market that leverages archival data will reach $9.9 billions globally in 2017 [90]. It is not uncommon for many businesses to retain archived collections for 10 to 15 years and in some industries the retention periods may be as long as 100 years or more [42]. A long period of data retention implies many versions may have to be maintained. The size growth of an archival dataset also becomes rapid with advancements in

cloud computing, content authoring, media sharing, and low-cost computer devices. Organizations also frequently backup their data. One of the largest archival databases is Internet Archive [64] which has collected and preserved more than 376 billion web pages in the last decade with tens of petabytes of data. Can backup files be used as an archive - all the information is in there, isn't it? It is; however there is a big difference between backup and archiving [63]. Backups are designed for recovery while archives require a sophisticated support for information retention, discovery and search. Discovery requirements grow every year, and backups simply cannot meet them. Archives are critical for some unanticipated legal or regulatory event and help discover details of information. Recently several companies have been developing an integrated backup and archiving solution. For example, EMC data domain [41], NetApp [82], HP [61] and IBM SmartCloud Archive [94] are developed to consolidate backup, archive, and disaster recovery with high-speed, inline deduplication. Efficient support for complex search and analysis on archival data is not extensively studied and there are limited The previous work on versioned data has studied the compression algorithms to identify shared data fragments among different versions [7, 19, 22, 34, 56, 57, 58, 60, 118].

Figure 2.1 depicts the original number of fragments divided by the number of unique fragments for a GitHub dataset we have tested when varying the number of versions per document. The result shows that the majority of fragments can be

removed in the search index since they are duplicates and this corroborates the importance of exploiting fragment-based compression.

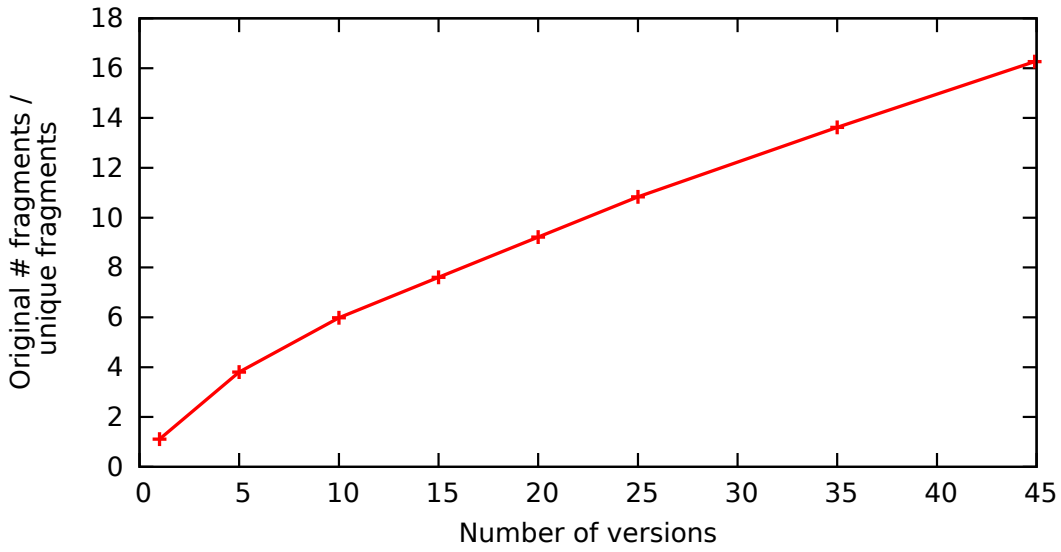


Figure 2.1: The ratio of the total number of fragments over the unique fragments in a GitHub dataset

Even though the index space for a versioned data collection can be compressed dramatically through fragmentation and deduplication, it is still time consuming to search the full index structure because a search procedure still has to deal with a large number of documents with many versions.

A two-phase approach [56, 89] has been proposed to find top results first using a non-positional index, which represents a tradeoff of relevance and search cost, and then rerank the selected top results with a positional index. The weakness of

the existing work on versioned data is that still there is a large number of versions to go through in Phase 1 even without a positional index.

This chapter is focused on processing conjunctive keyword queries on versioned datasets and our key idea is to extend the concept of cluster-based retrieval [4, 73, 75, 106] for representative-guided two-phase search and develop a per-cluster hybrid index to localize data access at Phase 2. Using representatives of document versions with full positional information reduces the number of top clusters needed to retain a good relevancy. The tradeoff is that Phase 2 requires memory caching of index or the use of solid state drives (SSD). To speedup Phase 2 search, we develop hybrid per-cluster indexing with adaptive traversal of forward and inverted structure. Our evaluation shows that the proposed scheme is up-to about 4x as fast as the two-phase approach [56] when the search index is available from memory or from an SSD.

The rest of this chapter is organized as follows. Next, we describe the background information and related work. Section 2.3 discusses design considerations in adopting cluster-based retrieval for searching versioned documents with representative guidance. Section 2.4 discusses the hybrid indexing and search options for fast Phase 2 query processing. Section 2.5 presents our experimental results on three real datasets which shows the accuracy and efficiency of the proposed techniques. Finally, Section 2.6 concludes the chapter.

2.2 Background and Related Work

In this section, we will provide background and related work. Firstly, we will discuss multiversion data index, which is on how to create index on archival data. Specifically, we will discuss non-positional index and positional index. Then, traditional data search and multiversion data search will be discussed, which is about how to retrieve search results on archival data. Fragment-based index will also be introduced here, which is the data structure we will work on in this chapter. Finally, we provide background information on documents clustering because it is highly associated with our cluster-based index.

2.2.1 Multiversion Data Index

The key data structure of document search is the *inverted index*. [121] is a detailed survey of related research work on inverted indexing. Given a data collection with n documents, the non-positional inverted index stores a posting for each term which is a list of document IDs that contain this word. When each posting also contains the position information of a term in each appeared document, this inverted index is called *positional inverted index*. Otherwise, it is called a *non-positional inverted index*. The index can also contain other weight information such as term frequency and document frequency. Here in our work,

we will use positional index which includes both frequency and offset for each word.

Since there is such high redundancy among versioned documents and such a large volume of data, compression is necessary for efficient index storage. For example, our experiment shows that the redundancy ratio can be more than 99.3% for large archival data such as the Linux kernel code on GitHub with 439 versions. Thus there is an up to 143:1 or more storage reduction with the deployment of advanced compression techniques.

Recent researches on multiversion data indexing can be divided into two categories: non-positional indexing and positional indexing. We will discuss about them separately as below.

Non-positional multiversion data indexing: There are a number of previous research work on non-positional index which only includes non-positional information. Comparing positional information, non-positional information indicates if a text word appears in a document or not while the positional information further captures the location of text words appeared in a document. The use of positional information is critical when query relevancy is sensitive to the proximity of query words matched in a document, but it incurs significant search cost. Non-positional indexing for searching versioned data is studied in [7, 19, 22, 34, 57, 58, 60]. Numbers in the index are compressed fur-

ther [6, 92, 113, 122]. One approach for the versioned compact index by Broder et al. [22] considers content sharing patterns among versioned documents with a tree structure and this work is further extended by Herscovici et al. [60] based on multiple sequence alignment. While this work does not address the positional information, some of their ideas can be leveraged in our work in computing the posting intersection within each document version cluster.

Another approach by Claude et al. is based on run-length, Lempel-Ziv or grammar-based compression with self-indexing [34] and the grammar-based compression such as re-pairing is also used for the document listing problem which searches substrings or phrases in versioned documents [35, 45, 50]. Most of these methods use index compression algorithms like OPT-PDF in [114], PForDelta in [59, 123] and so on [6, 93, 110].

Positional multiversion data indexing: Positional versioned data indexing is studied in [118, 34, 56]. The work of Zhang and Suel [118] uses the content-dependent partitioning method such as Winnowing [91] and 2MIN [101] to divide a versioned document into fragments and then each unique fragment after duplicate detection is only indexed once. The above partitioning technique is related to landmark-based indexing proposed by Lim et al. [74] for efficient index update when document content is changed. Duplicate detection is done by computing a

hash value for each fragment and comparing it with those of the already indexed fragments.

In [118], the idea is that firstly using a content-dependent string partitioning method to partition a whole document into several substrings; then, positional index is built on each fragment instead of the whole document. Using the stored information related between fragments and documents, the algorithm can easily infer the docID, frequency and offsets of a given term. The benefit of partitioning a whole document into substrings is that for multiversion data, documents between versions have lots of content overlapping. Therefore, a large number of substrings can be reused by many documents/versions, which saves both indexing space and time. As to the content-dependent string partitioning method, there are a number of related work [91, 62, 72, 81, 96, 97, 101] in OS, networking and many other fields. In [118], the author uses the Winnowing approach in [91] and 2MIN method [101] is used in [56] to do the content-based partition. The advantage of these partition methods is that in a large probability, a random insertion or deletion in the document will only affect one fragment. Thus, there will be a large number of common substrings between versions. An improved content partitioning and compression method is discussed in [56, 34].

A later work in [56] describes a complete framework for full-text positional indexing. Inheriting from [118], they develop several improved partition algorithms

by exploiting knowledge of the edit history of a document. The results show significant improvement compared to previous work.

On the other hand, the related work in [34] uses different methods such as run-length, Lempel-Ziv or grammar-based compression to reduce positional index size and indexing time.

2.2.2 Traditional Search

User experience is largely dependent on searching accuracy and the overall time consumed in the searching phase. Compared to the indexing phase which is offline, the online searching phase plays an important part in user experience. The searching problem is that given a query q=$\{t_0, t_1, ..., t_n\}$ which is composed of n terms, how to return the top m documents which are most relevant to the query. A function $F(d, q)$ which accepts one query q and one document d at a time and returns a relevance score is called a *ranking function*. Ranking functions vary from simple ones like BM25, Cosine, Okapi to some complex learning models such as AdaRank, Rankboost, and LambdaMART. However, no matter what ranking function is, it needs to use information from inverted index such as docID, frequency and positions. We refer to [121] as an overview of traditional searching methods.

Simple ranking functions like Cosine and BM25 is based on non-positional inverted index. Only information such as docID, term frequency and document length is needed to compute ranking score. For each document, after the score of each term is calculated, an intersection step will combine all the scores of all terms in the query and provide the final score. An early work [71] presents a DAAT algorithm to do fast intersection. However, it is proved that considering positional information in the searching phase, more accuracy ranking results can be achieved [79]. Thus, in this work, we develop our searching algorithm based on positional index.

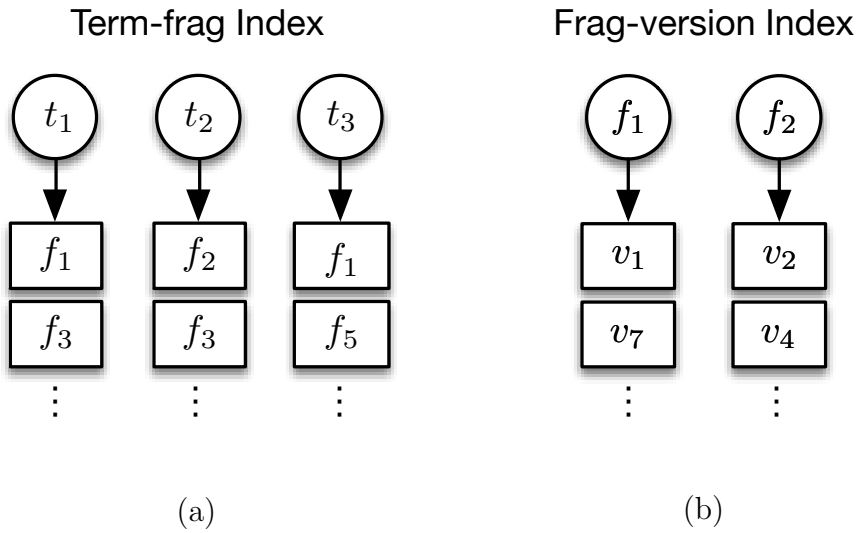
2.2.3 Multiversion Data Search

According to our investigation, although there are several recent papers on range query and query operations on versioned data [5, 54, 104], only a few amount of work is about standard ranking problem in multiversion positional data [89, 118, 56].

We adopt the fragment-based redundant content compression [118, 56] because fragments explicitly capture positional information and also because the work by He et al. [56] shows that it has a higher compression ratio than that of [34]. Grammar-based compression techniques [34, 50, 45, 35] have been used for phrase queries and we are more interested in exploiting more general positional infor-

mation. It is possible to adopt some of such techniques in version cluster index compression and this can be considered in the future work. While the tree-based compression or sequence alignment technique for content sharing [22, 60] does not address the positional information, some of their ideas are leveraged in our work for computing the posting intersection within each document version cluster.

We illustrate the fragment concept in more details as follows. Once a page is represented by a set of fragments instead of terms directly, the inverted index contains fragments in its compressed data layer. The posting for a term is a set of fragments instead of a page list and there is another data structure that maps a fragment to a set of page IDs that use this fragment. This data structure is caused *fragment-page reuse* table [118]. Noted that the above index structure can be augmented with term frequency information. Figure 2.2(a) is an example of the term-to-fragment index, following the compression scheme in [118, 56]. Each document version is divided into a set of fragments. The inverted index shows a list of fragment IDs that contain a term represented by a term ID. In this example, term "t1" is in fragments f1, f3, and others. Figure 2.2(b) shows an example of the fragment-to-version reuse table which is the list of page IDs that use this fragment. In this example, fragment f1 is used by document versions v1, v7, and so on.



(a) (b)

Figure 2.2: (a) Term-fragment inverted index. (b) Fragment-version reuse table that lists versions sharing each fragment.

The above compression for versioned data does make inverted indices more complex and its data layout increases the cost of data traversal during document matching. In [118], the search algorithm first identifies a set of fragments that contain a query term and then uses the fragment-page reuse table to locate versions containing each term and the offset of each occurrence within the versions. The intersection for processing multiple conjunctive query words is arranged accordingly and those version pages which contain all the query terms are scored and ranked. The advantage of using the compact page-fragment-term structure is a substantially smaller index compared to the standard document-term inverted index. On the other hand, the intersection of multiple postings takes more resources by going

through the term-fragment-page mapping. For example the tradition intersection algorithm can take advantges of sorted document IDs by employing skipper or other posting auxiliary data structure while the fragment index cannot. Comparing searching on the index which is directly built on full versions of all pages, this method is much faster because the index is much smaller by exploiting content redundancy. However, this algorithm still cannot satisfy user's needs because the job of searching on position data itself is too expensive.

Instead of fetching on all the data D, [89] proposes a method of using a non-positional index in the first phase and then search on the positional index of a much smaller dataset D_{small}. In the first step, a simple ranking function is used to retrieve a top-k' results. After that, a complex ranking method is applied but only on the positional data of the top-k' results from the previous step. Finally, the top-k results are generated.

The work by He and Suel [56] extends the fragment-guided two-phrase search based on [89, 118]. Phase 1 uses non-position index while Phase 2 contains detailed position information. In their first phase, they implemented a two-level index structure called 2R-MSA from [58]. Runtime query processing searches non-positional inverted index first to retrieve the top K results and then re-rank these top K results by accessing the full positional index of these versions. The above two-phase search is motivated by the earlier research on cluster-based re-

trieval [4, 106, 75] for non-versioned documents. Using a non-positional index at Phase 1 is also motivated by relevancy studies [9, 10] and the argument is that having a sufficiently large number of top K at Phase 1 without positional information can still deliver a good relevancy. Still there is a large number of versions to go through even without positional index.

2.2.4 Documents Clustering

Since our method uses document clustering to choose representative, at the last part of this section we briefly discuss some recent work on documents clustering here.

Document ranking using cluster-based retrieval and language models is studied by Liu and Croft [75], and cluster ranking is addressed by Kurland and Krikon [73]. Index optimization for cluster-based retrieval for traditional disk storage is studied in Altingovde et al. [4]. We revisit the cluster-based retrieval techniques as the storage seek overhead of random access has been reduced significantly with today's SSDs. The recent work for searching non-versioned data in Bai et al. [18] considers the use of flash-memory drives to store the per-document forward index. We exploit adaptive use of hybrid cluster-specific index structure.

2.3 Cluster-based Retrieval with Representatives

Deduplication reduces the storage demand significantly in archiving raw documents and creates opportunities for highly efficient versioned data search since there are highly repetitive content among different versions of documents. To maintain the efficiency of search, our proposed algorithm continue uses a two phase design while using the positional index in both phases to improve the relevance for conjunctive queries. In this section, we first discuss our design considerations. Then, we talk about how to choose cluster representatives in detail.

2.3.1 Design Considerations

Our objective is to develop a faster search scheme with much faster query processing time, especially when there is a large number of versions. Our design considerations are discussed as follows.

- Since versions of documents have highly repetitive content, most likely there exists a version or a composed version that could capture the majority of text features for many versions of each document. Our work is motivated by the cluster-based retrieval [106, 75, 73, 4] which was proposed to rank non-versioned data by exploiting document similarity in clustered results.

For versioned datasets, exploiting document similarity can be more important to reduce search time because of high similarity among versions of documents. Thus we adopt the concept of the cluster-based retrieval and consider versions of a document as a group. We compose a representative which captures positional and non-positional information for each version cluster to facilitate cluster ranking. A phased search can start with a set of representatives instead of the entire document collection. The clustered index with representative-guided search can quickly narrow the search scope and results in a big reduction of search time, even there are a large number of versions with similar content. Since the number of representatives is modest after removing the versioning effect, it is not necessary to avoid positional information in Phase 1 index and save index space cost. This approach can also improve the diversity of ranking results so that results from one document with many versions will not dominate the entire ranking.

- There is a storage access cost to retrieve cluster-specific information for each selected cluster at Phase 2. For non-versioned data, the work in [4] studies a per-term cluster posting so that the number of disk seeks is controlled as the number of query words. We can extend this work to build per-term index structure for versioned documents, but the repetition of cluster information from one term to another becomes extremely large. Consider

the SSD storage with low seek time (e.g. 0.1 ms) is getting popular, we can afford to access a modest number of clusters dynamically.

- Fragment-based index compresses versioned data significantly while runtime index traversal becomes slow in order to fetch positional information access during Phase 2. This gives opportunities for optimization and we will present a hybrid indexing solution that combines the strength of forward and inverted index. This follow-up phase can be conducted to identify the individual document versions with a high rank within each selected group. Noted that the two-phase search for a general non-versioned dataset [9, 10] considers the tradeoff of performance and relevancy by deploying non-positional index in the first phase of search. While it is true, choosing a large top k number in first phase exposes most of relevant documents for Phase 2 search. But using popular words in a query can match most of documents in a collection, as a result, Phase 1 narrowing without using positional index becomes meaningless for a large data collection when a very large k is selected. The relevancy impact without using positional index in Phase 1 is more serve for versioned datasets because of highly repetitive content. The representative-guided approach with a small core index supports more flexibility for obtaining better relevancy.

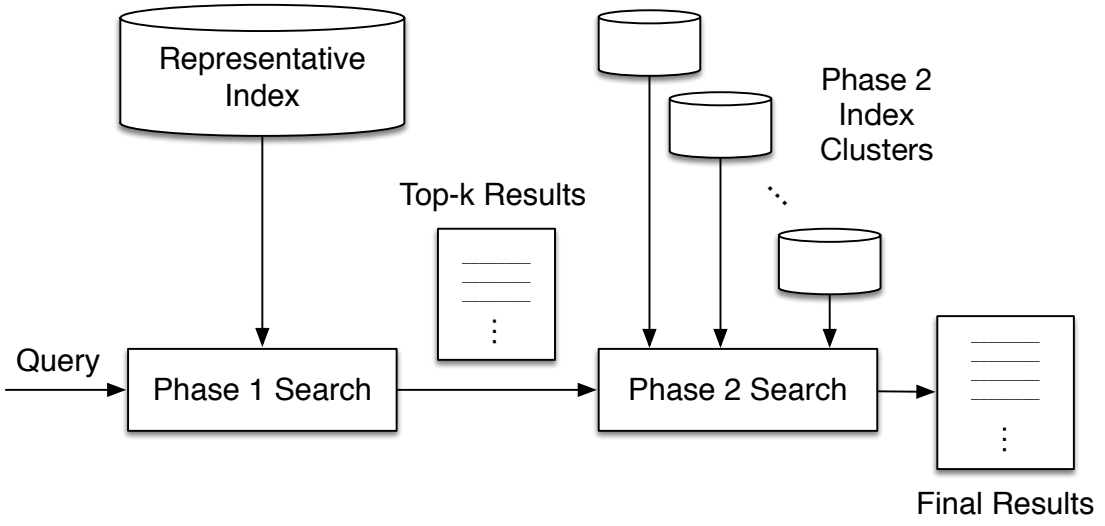


Figure 2.3: Representative-guided search workflow for query processing

- Clustering of similar content and the construction of a representative for a group deserves careful research studies. We would like this representative document to catch terms that appear in different versions of the same document. We will elaborate this in the next subsection in detail.

The two-phase query processing workflow is depicted in Figure 2.3. The first phase is going through the representative index and retrieving the top k representatives. The second phase is searching on the related index clusters and aggregating the matched results from these clusters. Both phases use positional information. Algorithm 1 lists the search procedure. Since each document has many versions, to ensure diversified results, we restrict the number of versions

shown for each document in the final result list and a user can select a document and view more versions of the same document if interested.

```
1 Phase 1
2 Find top k document clusters that match queries.
3 Phase 2
4 Fetch k cluster-specific index.
5 for each selected cluster do
6     Find document versions that contain conjunctive query terms.
7     Fetch positional and non-positional information to form rank features
        and scores.
8 end
9 Select top document versions from all clusters based on ranking scores
```

Algorithm 1: Two-phase search with cluster-base retrieval for versioned documents.

Representative and final result ranking. For a standard text document search system, there are many ranking signals that can be extracted or characterized for each matched document in processing a query [3, 88, 115]. Those signals need to be aggregated, and their weights can be imposed empirically or derived using a machine learning method (e.g. LambdaMART [111]). We follow the previous work in text ranking with proximity (e.g. [17, 25, 68, 98, 100, 120,

119, 111, 116, 112, 75, 73]). Our study focuses on the construction of index that can provide basic ranking features which can be used to compose scores for these ranking schemes. The text features for ranking include BM25, query word span, query word co-occurrence, and query word minimum distance based on the previous work in text document ranking with proximity [17, 25, 68, 98, 100, 120, 119]. Weights of non-positional text features and proximity features are differentiated by where matched query words appear in the fields of a document like title or body. Since each document has many versions, to ensure diversified results, we restrict the number of results shown for each document version group in the final result listing and a user can select a group and view more versions of the same document if interested.

2.3.2 Cluster Representatives

To aid the top k ranking of clusters at Phase 1, we compose a representative for each cluster to provide essential ranking features. The representative for each version group is a superdocument that is artificially composed from a specific version of a document. Our objective here is to create a superdocument that includes terms that appear in all versions of a document while capturing the majority of distance relationship among these terms.

The position information of a representative superdocument is built based on the longest version. More specifically, let s_i be the super document for document d_i with m versions. Denote these versions as $v_{i,1}$, $v_{i,2}$ ... $v_{i,m}$. Let $v_{i,l}$ be the longest version for d_i. Let $T(s_i)$ be the set of terms included in super document s_i. Let $Pos(t, s_i)$ be a position information set for term t in $T(s_i)$. The index for representatives with the longest version is defined as

$$t \in T(s_i) \text{ if } \exists j, 1 \leq j \leq m \text{ and } t \in T(v_{i,j}).$$

$$Pos(t, s_i) = \begin{cases} Pos(t, v_{i,l}) & \text{if } t \in T(v_{i,l}) \\ \infty & \text{if } t \in T(s_i) - T(v_{i,l}). \end{cases}$$

Namely for the extra terms in set "$T(s_i) - T(v_{i,l})$" added to the superdocument we will treat the appearance of these terms in an unknown position with the ∞ offset value. In the ranking process for document retrieval, a position with the ∞ value will not contribute to the position related weight in the final score aggregation. We can also consider other options to form a superdocument, for example, based on the latest version.

A representative contains terms from all versions, and thus there is a false positive error occurred during Phase 1 keyword matching. Some positional information is not precise because some sentences are obtained from various versions. Based on our experiment results, we observe that representatives do not introduce significantly more new words. The false positive error introduced in Phase 1 is

fairly small. In the second phase, the search algorithm makes an accurate assessment of term appearance in all versions for the matched representatives from Phase 1 to correct errors if there is one.

The above method tries to capture positional information as much as possible from all versions while using a reasonable amount of space. There is a tradeoff here that it cannot capture all positional information. A future study includes the consideration of clustering of similar versions from the same or different documents. We expect a good similarity clustering with proper representative construction can improve the accuracy.

During offline data processing, each document version is divided into fragments using a content partitioning algorithm called TTTD (Two Thresholds Two divisors) based on the work of [43, 80], which is a similar but faster method comparing the ones used in [118, 56]. Since content redundancy among representatives is modest or less significant, we can use the traditional index structure for representatives without using fragments for simplicity and efficiency.

Note that the meta information for each version of a document also contains a timestamp. The time data will be useful for user queries containing temporal constraints. The offline index may also be partitioned based on a coarse-grained time interval to optimize the index search if that matches the users' query pattern. Studies on versioned data search with a time range are in [19, 55, 105]. This work

focuses on conjunctive query processing and the time range can be added as a filter in the search process.

2.4 Hybrid per-cluster Indexing and Traversal

Phase 2 query processing needs to identify document versions from selected clusters that really match required keywords, and compute a ranking score. It would be more expensive to compute scores for all versions and then filter out those that do not contain all conjunctive keywords. We discuss how to gather basic feature data from the index for intersection and scoring below. The basic feature data includes the positional information of each term in a version and frequency information can be derived from this process if it is not explicitly stored.

As we adopt the fragment-based compression in the per-cluster index with the positional information, one option is that an intersection algorithm uses the term-fragment list and looks up a fragment-version reuse table dynamically to determine if all conjunctive query words included in an version. From our experiments, we find the dynamic conversion is very time-consuming. The versions derived from the fragment-version reuse table do not follow a sorted relationship. It is much faster to construct the term-version posting first before performing the intersection operation and how to optimize the traversal of index data structure is the key to accomplish a low search time.

2.4.1 Indexing and Search Options for Phase 2

We discuss two indexing options first with different time and space tradeoffs in considering the traversal of cluster-specific inverted index or forward index and then design a hybrid option.

- Option A: Each posting list of a term in a version cluster index is composed of fragments and the positions of this term in each fragment. The runtime search process extracts this list from the term-fragment index and traverses a per-cluster fragment-version reuse table to reconstruct a term-version posting with positions for each term. This derived term-version posting includes the positional and frequency information and after that, a multi-keyword intersection is conducted using such postings.

  In detail, we extract the term-posting with positional information from the inverted term-fragment index and fragment-version mapping. There is a performance challenge in utilizing the fragment-based index in each document group for posting interestion. From the term-fragment index, we can get a list of fragments which contain the query term. After a set of document versions that contain the matched query words is derived, each group uses the term-fragment index to obtain a list of fragments that contain a

word. Then we use the fragment-version mapping to obtain the positional information for these matched versions.

For example with query "arabic number", we will get two fragment lists: one for "arabic" and the other for "number". Then by checking the fragment-version reuse table, we reconstruct a term-version positing with positions for both "arabic" and "number". After that, a multi-keyword intersection is conducted and ranking is done as the last step to return the final top k results.

- Option B: Compared to Option A, this option adds the extra storage space overhead to explicitly store the term-version posting of each term, and a version-fragment forward index, but it does not need to maintain the local fragment-version reuse table. As the version posting of each term without positional information is available in advance, the intersection of term postings can be conducted quickly first without a need to dynamically reconstruct such postings. Once a set of matched versions is derived through the intersection, the query-time process derives positional information by finding all fragments included in these versions through the forward index, and then by using a binary search on a term-fragment posting to extract the positional information of each term at each document version.

For example, given query "Marine Science", for each version cluster derived from Phase 1, we directly use the term-version postings of "Marine" and "Science" to conduct an intersection. Assume v_1 and v_2 contain both terms, then we use the version-fragment forward index to identify possible fragments and their positions in v_1 and v_2 that may contain each term. Finally a binary search using the term-fragment postings of "'Marine" and "Science" identifies the text positions in real fragments that truly contain these two words.

A detailed discussion of data structure choices of above two options is in Section 2.4.3 summarized in Table 2.1. There is a time and space tradeoff between Option A and Option B. Option B can be faster than Option A in many cases while it does use slightly more space. On the other hand, Option A can outperform Option B some time, for example, queries using rare terms. We model the time cost of Options A and B as follows.

$$TimeCost_A = k \cdot (\Gamma + \sum_{i=1}^{q} f_i \cdot \mu_i \cdot \rho_i \cdot \tau + \Pi), \text{ and}$$

$$TimeCost_B = k \cdot (\Gamma + \Pi + \sum_{i=1}^{q} m \cdot \gamma \cdot (\log(f_i) + \rho_i) \cdot \tau)$$

where

- k is the number of top clusters selected by Phase 1;

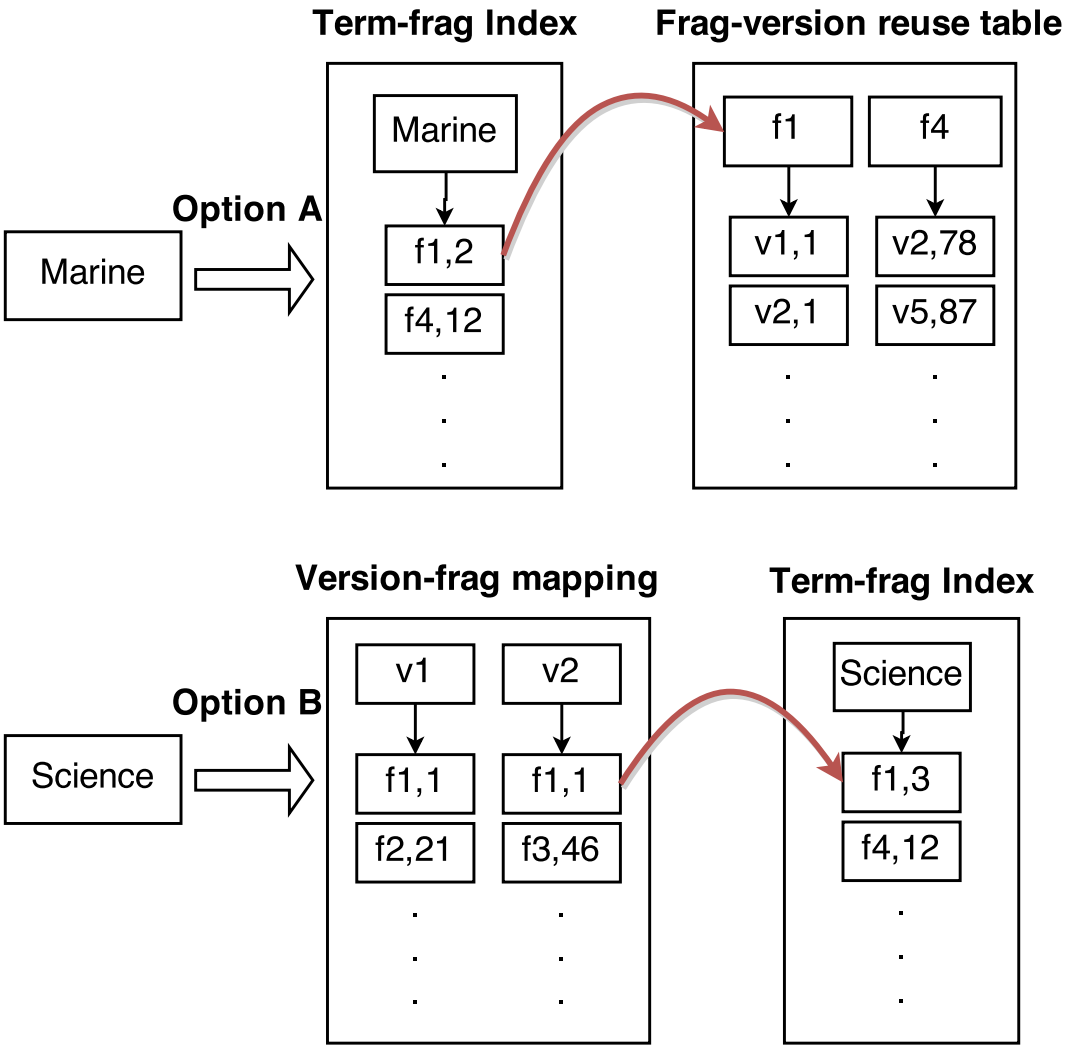


Figure 2.4: An example of data traversal in Option C that selects the Option A or Option B approach for each query word at each cluster.

- q is the number of query terms;
- f_i is the number of fragments for term t_i's posting at a cluster;
- m is the number of matched document versions after query intersection;
- Γ is the average cost of loading a cluster index to memory from a fast storage;
- Π is the average cost of posting intersection for conjunctive keywords;
- μ_i is the average number of document versions using a fragment that contains term t_i;
- ρ_i is the average number of positions of term t_i in one fragment;
- γ is the average number of fragments included in one version.
- τ is the average memory access time.

The index data for each cluster is organized separately and the runtime system loads the data k times for the selected top k clusters from a fast solid state drive. For our tested applications, the data size for each cluster to be load is less than 5KB and the per-cluster index I/O time Γ is about 0.28 milliseconds or less using SSDs. The overhead is not overwhelming when fetching the index data for top few hundred clusters. Π represents the cost of intersection and we discuss this part in next subsection.

The cost of rank score computation is not explicitly included in the above formula. Different ranking methods deploy different features which can incur a different amount of time [36, 98, 100, 25, 17, 119], while we expect such a cost is in an order subsumable under the above cost formula. (e.g. [21, 49, 47, 67, 111, 116, 112]). Our experiment uses a model of query word span [98] and frequency-based BM25 score and the calculation cost is proportional to the number of query word positions appeared at each version.

From the cost model of options A and B, one can infer that if a query term t_i is rare, and f_i is very small, this case favors the use of Option A. On the other hand, if t_i is a popular word, and f_i is very large, this case favors Option B. Theoretically, Option A and Option B give better performance under different scenarios depending on the value of the parameters, which differs for different datasets and different query terms. We are devising Option C that compares the relative cost ratio of Options A and B for extracting positional information and adaptively select one of them based on the characteristics of each term t_i searched in each cluster. The query-time work flow of Option C is summarized in Algorithm 2.

Figure 2.4 illustrates an example of Option C in processing the query "Marine Science". First, the intersection finds v1 and v2 contain "Marine" and "Science". For keyword "Marine", the threshold condition leads to the use of Option A traver-

```
1  Load the cluster index and accumulate statistics.
2  Perform the intersection of version postings.
3  for each query word t_i do
4      If f_i · μ_i · ρ_i ≤ m · γ · (log(f_i) + ρ_i),
5      Use the term-fragment posting of t_i to get fragments and term
         positions within fragments.
6      Convert into term positions within all versions by accessing the
         fragment reuse table.
7      Else
8      If needed, set L as a list of fragments for all matched versions.
9      Search t_i's fragment posting to find term positions for the fragments
         in L.
10     Derive term positions within matched versions.
11 end
```

Algorithm 2: Option C that extracts position and other meta information of the matched versions within each cluster.

sal. From the fragment posting of "Marine", a list of fragments are found and they correspond to versions v1, v2, v5 and so on. Only v1 and v2 are acceptable from the intersection results, then the position at v1 is computed as 3 and the position at v2 is computed as 78+12 which is 90. For keyword "Science", a different path is taken: from version-fragment mapping, v1 contains a list of fragments, such as f1 and f2. By doing a binary search on the Term-frag list of "Science", we know in version v1 only f1 contains "Science" and position 4 is derived at version v1. We do the same procedure with v2 and position 4 is derived at version v2.

It should be noted that Option C's decision path is cluster dependent. Namely this method selects Option A or B based on the difference of $f_i \cdot \mu_i \cdot \rho_i$ and $\cdot m \cdot \gamma(\log(f_i) + \rho_i)$ for each cluster. Thus for the same query word, one cluster can make a selection different from another cluster. The time cost of Option C can be modeled as

$$k \cdot (\Gamma + \Pi + \sum_{i=1}^{q} min(f_i \cdot \mu_i \cdot \rho_i, \quad m \cdot \gamma(\log(f_i) + \rho_i)) \cdot \tau)$$

which is usually smaller than that of Option A or Option B.

2.4.2 Term-version Posting Intersection

Term-version posting intersection without position information is used in Options B and C and we can leverage the ideas of the work in [22, 60]. When the number of versions is modest for each version cluster, the intersection to identify

versions of a document containing all keywords can be extremely fast by representing the posting using a bit operation [37]. For a large number of versions, the intersection with bit operations is less effective. Since versions of a document tend to be similar, when a word appears in one version, it appears in another version with a good probability unless this word is added to a version and is deleted shortly after a few versions. With this data characteristic in mind, a long bit vector can be represented succinctly by a small set of intervals. Each interval represents consecutive version numbers that contain this keyword.

When each version posting is represented by a set of intervals, the intersection algorithm of posting intervals from multiple words can be conducted by modifying the integer set intersection algorithm by Culpepper and Moffat [37] with a change in Golomb search to accommodate the interval boundary comparison. For interval list intersection of two words, assume one word has n_1 non-overlapping intervals and anther word has n_2 non-overlapping intervals where $n_1 < n_2$. The maximum number of new intervals produced as the result of intersection is $n_1 + n_2 - 1$. The time cost of this intersection is $O(n_1 \log \frac{n_1+n_2-1}{n_1})$. For q word intersection, the corresponding bit-vectors have $n_1, \cdots n_q$ intervals with $n_1 \leq \cdots \leq n_q$. The total number of subintervals produced can be high, but less than H where $H = min(V, \sum_{i=1}^{q} n_i - 1)$ and V is the total number of document versions in a cluster. The total time cost is bounded by $\Pi = O((q-1)H \log \frac{H}{n_1})$. Because $n_1, n_2, \cdots, n_q$

tend to be small in a cluster index, the time cost Π is not significant. Note that the hashing-based intersection [39] does not work well as the intersection of two intervals requires an inequality comparison of interval boundary values. Skip-based synchronization points [37] may be used with extra space while adding them has not given a visible improvement for this problem in our experiments.

2.4.3 Index Storage Layout and Cost

We discuss the data layout for the cluster specific index structures used in the above three options, and assess the space cost difference in an approximated manner. The sequence of numbers used in each data structure is further compressed by storing their delta gaps and using one of number compression methods with fast query-time decompression such as var-Byte, Simple-9, PforDelta, and Opt-PFD [92, 122, 117, 113]. Through experimentations, we find that a combination of Simple-9 [6] and var-Byte gives a competitive compression performance in our context.

We estimate the storage need of each index option approximately before applying the aforementioned number sequence compression. This gives a rough space assessment assuming the number compression brings is at a similar level of space reduction proportionally. Since small integer values typically use less space after number compression, and to improve the accuracy of the space cost estimation,

we associate the integer size as a coefficient. Document IDs and word IDs need a 4-byte integers in general, we assume that the version numbers of each document and position numbers need integers with 1 to 2 bytes. Also we perform the fragment ID localization under each cluster so that local fragment IDs may be ranged with 2 bytes or less. These two-byte numbers are dominating the cluster index and thus we estimate space cost approximately in terms of the number of two-byte short integers.

We will use the following additional symbols in addition to ones used in time cost analysis.

- W: the number of distinct words at a cluster.
- R: the number of fragments at a cluster.
- V: the version number at a cluster.
- μ: the average number of versions using a fragment at cluster.
- ψ: the average number of fragments per term at a cluster.
- β: the average size of the posting bitvector of a term discussed below.

Term-to-version posting bitvectors: The extra term-to-version posting for Options B and C records version IDs that contain a term without positional information. When V is not too large, an internal bit vector representation is

appropriate. When V becomes modestly large, we consider a hybrid compression as follows. Since versions of a document are similar, either many versions shared the same words or they have little in common for other words used in the cluster. The characteristic of a posting bit vector for our version dataset is that each vector either contains lots of 1's or lots of 0's. We can either use a few intervals to represent a bit vector or follow a hierarchical compression scheme from [46]. The root of a tree structure in [46] can use bit value 1 to represent a large consecutive number of 1's when 1 is dominating a bit vector. Otherwise the root uses 0 to represent a large consecutive number of 0's. The space for bit vectors of W terms is $\beta \cdot W$.

Term-to-fragment posting: This posting contains a fragment list and term positions at each fragment. Because most of terms appear in a fragment once, we follow the idea from [118, 56] to store a sequence of fragment and position pairs. The sequence of numbers for each entry in this index is: (term ID, meta information, fragment ID, position, fragment ID, position, $\cdots$). The meta information represents the number of pairs and other control flags. The size of term-to-fragment index is $(5 + 4\psi) \cdot W$.

Local fragment-to-version reuse table: Following the same strategy for a term-to-fragment posting, we record the number of version and position pairs in

the sequence: (fragment ID, meta information, version ID, position, version ID, position, ...). Total size is $(3 + 4\mu) \cdot R$.

Version-to-fragment mapping: Similarly, the number sequence of an entry in this mapping is: (version ID, meta information, fragment ID, position, fragment ID, position, $\cdots$). The total size of the forward index is $(3 + 4\gamma) \cdot V$.

Considering the total number of fragment occurrences in a local reuse table or in a forward mapping index across all the document versions remains a constant in a cluster, we can show that $\mu R = \gamma V$. Thus, the version-fragment mapping and fragment-version mapping have very similar size.

Following data structure choices used in each option, the total space cost of each option for each cluster before number compression is estimated as:

$$SpaceCost_A \approx ((5 + 4\psi)W + 3R + 4\gamma V),$$

$$SpaceCost_B \approx ((\beta + 5 + 4\psi)W + 3V + 4\gamma V),$$

$$SpaceCost_C \approx ((\beta + 5 + 4\psi)W + 3R + 3V + 8\gamma V).$$

There is some additional space need for meta information such as version document length, which is less significant. The difference ratio between Options A and B is approximately

$$\frac{\beta W + 3V - 3R}{(5 + 4\psi)W + 3V + 4\gamma V}.$$

The difference ratio between Options B and C is approximately

$$\frac{4\gamma V + 3R}{(\beta + 5 + 4\psi)W + 3R + 4V + 8\gamma V}.$$

Note that $R << \gamma V$ assuming high redundancy among versions of the same document. From the experiment data we have tested, $\psi \approx 2$ and γ is in between 13 and 20. When $V << W$, $\beta \approx 3$ and the difference between A and B is about $\frac{\beta}{5+4\psi}$ which is 23% and the difference between B and C is small. When $V >> W$, the difference between A and B is small and the difference between B and C is close to 50%.

Table 2.1 summaries the data structure choices in three options. Option C is faster than Option A or Option B while incurring a modest increase in storage cost.

Per-cluster data structure	Option A	Option B	Option C
Posting bitvectors	no	yes	yes
Term-to-frag. index	yes	yes	yes
Frag.-to-version reuse table	yes	no	yes
Version-to-frag. mapping	no	yes	yes

Table 2.1: Data structure choices for three options.

How much does storage space overhead increase by building separate index for each cluster? We compare space cost difference between Option C and a corresponding global index with no cluster separation. Since there are full or partial duplicates among documents, the separation of index by clusters has a space disadvantage that some of term IDs appear redundantly in the local term-to-fragment postings. Some of fragment IDs also appear redundantly in both term-to-fragment postings and the local reuse tables. On the other hand, the fragment IDs after localization and version IDs in a cluster index uses less number of bytes compared to a global index and we assume a 1:2 ratio for fragment IDs, version IDs but still 1:1 ratio for positions. The global index space is modeled

$$SpaceCost_{global} \approx \beta Wn + 5M + 6\delta\psi Wn + (6\delta R + 6V + 12\gamma V)n$$

where n is the number of documents; M is the number of distinct words globally; δ is the ratio of fragments which are not shared among documents. In our test datasets, $M \approx 0.05nW$ and δ is between 0.7 and 1. V, W, and R have the same meaning as before, except it is an average number per cluster. Thus the space difference ratio between global index and option C is

$$\frac{[5(1 - \frac{M}{nW}) + (4 - 6\delta)\psi] \cdot W + (3 - 6\delta)R - 3V - 4\gamma V}{(\beta + 5 + 4\psi)W + 3R + 3V + 8\gamma V}.$$

When $V >> W$, the difference ratio is about $\frac{-3-4\gamma}{3+8\gamma}$ which is close to -50%. Namely the global index is about 50% larger. When $V << W$, the difference is about $\frac{5+(4-6\delta)\psi}{\beta+5+4\psi}$ and thus the global index is up-to 28.8% smaller.

2.5 Evaluations

2.5.1 Datasets and Experiment Settings

Objectives. Our evaluation objectives are:

- Demonstrate the benefits of searching versioned data with cluster-based retrieval in terms of query search time difference;
- Compare the index options in building the Phase 2 index in terms of time and space cost;
- Assess the impact of relevance for using a core representative index with positional information.

Settings. In this context, we study the impact of using different representatives. We have developed a prototype to implement the proposed approach with C++. Code is compiled with G++ using optimization flag -O1. Experiments are conducted on a Linux CentOS 6.6 server with 8 cores of 3.1GHz AMD Bulldozer FX8120, 16GB memory, Kingston HyperX 3K 240GB SSD and a 1TB Western

Digital Caviar Black hard disk drive (HDD) with 7200 RPM. All experiments store the index data in the SSD except Table 2.3 which compares time performance when the search index is stored in HDD and in SSD.

Datasets. Since there are no standard benchmarks for versioned search, we have crawled the following three versioned datasets for evaluation. Our datasets are available upon requests for other researchers.

- The first one is from Wikipedia (called Wiki), which consists of 3.8 million articles and on average each distinct document contains 13 versions. There are $315,673$ distinct documents. The versions are archived in a monthly crawled gap from April 2013 to April 2014. The total size of all 13 version raw Wiki data is about 100GB. On average each document version has 1.9 KB and each fragment has 156 bytes. There are $5,160,703$ distinct fragments after deduplication.

- The second dataset is a web dataset (called Web) of 5 million pages and there are 20 versions per document on average. There are $252,285$ distinct documents. The dataset was crawled in 2014 from two university domains. The total size of raw web pages is 120 GB. After removal of HTML tags, each document has 3.1KB and each fragment has 191 bytes on average. There are $4,138,852$ distinct fragments after deduplication.

- The last dataset is from GitHub (called GitHub) with 5 million versioned documents in total. Those are Linux code documents dated from April 2005 to April 2014 with 439 versions per document. There are $19,548$ distinct documents. Searchable text is extracted from the source code using function names and embedded comments, and each document corresponds to one source file. The total size of 439 versions of raw data is approximately 86GB. After pre-processing, each document has 3.35 KB and each fragment has 167 bytes on average. There are $714,793$ distinct fragments after deduplication.

Ranking function. Following the previous text ranking techniques (e.g. [25, 68, 100, 119]), the text feature in our implementation leveraging non-positional information is standard Okapi BM25, differentiated by where they appear in the fields of a document such as title and body. We also use a text proximity feature that leverages positional information: the length of a minimum text *span* [98] to cover query words at each field, scaled by the percentage of query words covered. The overall ranking score linearly combines weighted BM25 and proximity features.

We compare our Representative-guided Two-phase Search (RTP) with the following approaches:

- One phase search (OP). The implementation is based on [118] using the positional index with fragments. This is essentially the same as Phase 1 of

RTP except that OP searches the entire fragment-based index. This is the slowest algorithm but it has no accuracy loss.

- Two phase search (TP). This is based on [56] and the Phase 1 implementation ranks top-K results with non-positional index. Phase 2 re-ranks the selected top K results using the positional index with fragments. For RTP, we select the number of representatives k ranked at Phase 1 between 10 to 100 to produce good final top 10 results as an answer.

For TP, K is chosen to be $K = k * V$ so that there are enough good results selected from each cluster at Phase 1, which allows re-ranking at Phase 2 to produce relevant results competitive to RTP. The three datasets have different V values, affecting K values and performance difference of TP and RTP in addition to relevancy. We will also show the relevance results of RTP and TP with different k and K values.

For search time measurement, we report the average query processing time for each query set through 25 runs when excluding or including the index load overhead from a disk drive. We have generated 500 synthetic queries for each dataset with query length distribution following a query log study in [8] for the purpose of performance assessment.

Regarding to index disk loading time, we put Phase 2's index in our SSD disk because of the number of random seeks cannot be neglected in reading different

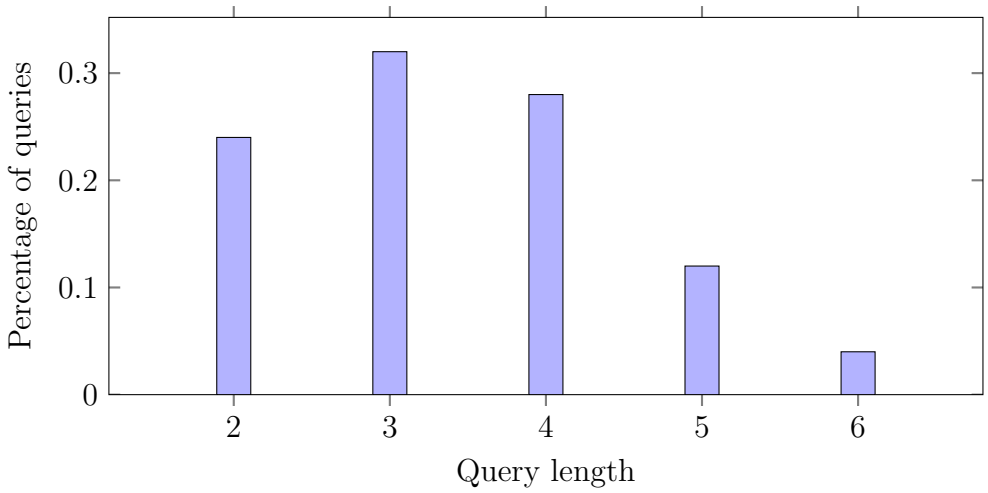


Figure 2.5: Percentage of queries for different query length

clusters' indexes. Several recent studies [102, 107] have given the comparing results of SSD and HDD random seek time. Thus, using an SSD disk for Phase 2 index can give a large benefit. As to Phase 1's index, it is just a traditional search index. There is no specific requirements of using SSD for it.

Our scheme is designed for handling conjunctive queries with two or more words and the relevance of RTP for single-word queries is about the same as TP. Thus our evaluation is focused on such queries.The length distribution of these queries follows is depicted in Figure 2.5.

For relevance, the standard relevance metric for document search is NDCG [66]. Since there are many versions for each document with similar content and some of which can be considered as near duplicates, showing too many results per document would affect result diversity [83]. Thus we restrict the displaying of top

results and only show v versions per document. A user can request for accessing more versions from a selected document when needed. We expect v to be 1 or 2 at most in practice. In our evaluation, we use $v = 1$ and collect the NDCG value at top 10 positions. We call the modified NDCG score as vNDCG1@10.

2.5.2 A Comparison on Overall Search Time

Time (ms)		OP	TP	RTP
Web	Phase1	5899	34.56	16.26
	Phase2	0	115.5	21.36
	Total	5899	150.0	37.62
Wiki	Phase1	1047	5.378	3.868
	Phase2	0	119.9	28.81
	Total	1047	125.3	32.68
GitHub	Phase1	5846	46.50	2.394
	Phase2	0	539.0	139.7
	Total	5846	585.5	142.1

Table 2.2: Query processing time in milliseconds when the search index is preloaded to memory.

For fair comparison, to prepare same number of candidate pages in Phase 2 for TP and RTP, we set $K = k \cdot V$. We use $K = 2000$ and $k = 100$ for the Web

Time (ms)		OP	TP	RTP
Web	SSD	5901	153.4	67.28
	HDD	5950	252	938.6
Wiki	SSD	1049	128.7	55.92
	HDD	1098	227.3	738.2
GitHub	SSD	5848	588.9	147.4
	HDD	5897	687.5	303.6

Table 2.3: Query processing time in milliseconds including the index load cost from an SSD or HDD.

with $V = 20$, $K = 1000$ and $k = 77$ for Wiki with $V = 13$, and $K = 5000$ and $k = 12$ for GitHub with $V = 439$. Table 2.2 lists the time cost of RTP, OP and TP in milliseconds when the search index is kept in memory. OP that navigates all versions of documents is much more time consuming than TP and RTP. For the Web dataset, OP takes 156x more time than RTP. For the Wiki dataset, OP takes 31x more time than RTP. For the GitHub dataset, OP takes 40x more time than RTP.

In terms of TP vs. RTP, for GitHub with a large number of versions, RTP is about 4.12x as fast as TP. On the other hand, for the Web and Wiki datasets with a modest number of versions, RTP is about 3.99x as fast as TP on Web data

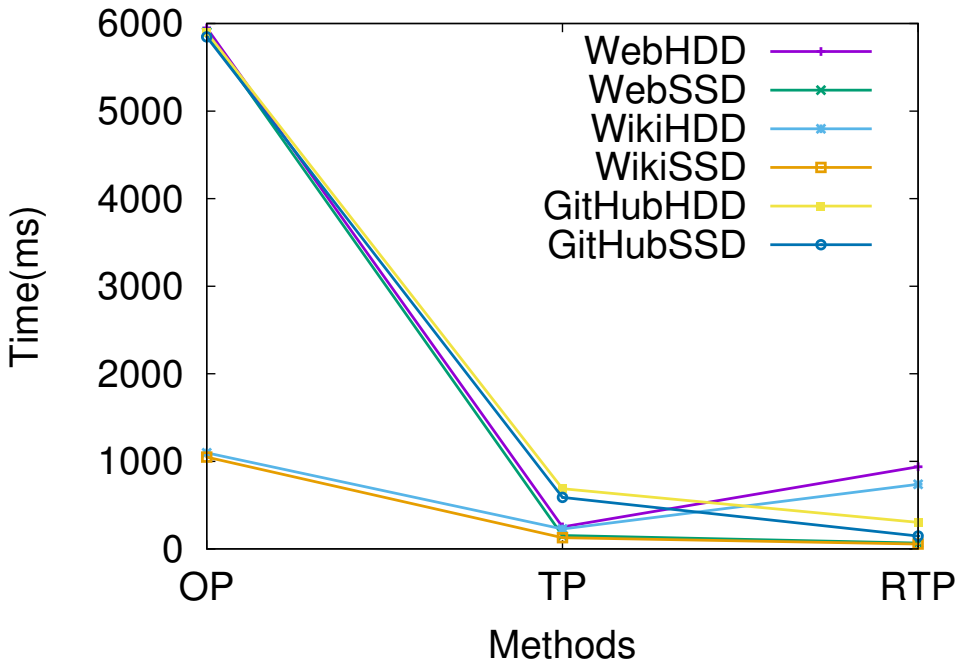


Figure 2.6: Query processing time of different search methods when the index is on SSD or HDD.

and 3.83x on Wiki data. RTP exhibits a fast performance in dealing with a large number of versions. This is because the high search scope reduction using the representative index has more efficiency benefits as the number of versions per document increases. In general, RTP has better efficiency than TP algorithm no matter the dataset has large number of versions or small number of versions.

Table 2.2 also lists the cost distribution of Phase 1 and Phase 2 time in detail. RTP's Phase 1 is faster than that of TP because RTP's Phase 1 search scope focused on representatives is much smaller while Phase 1 of TP searches the index

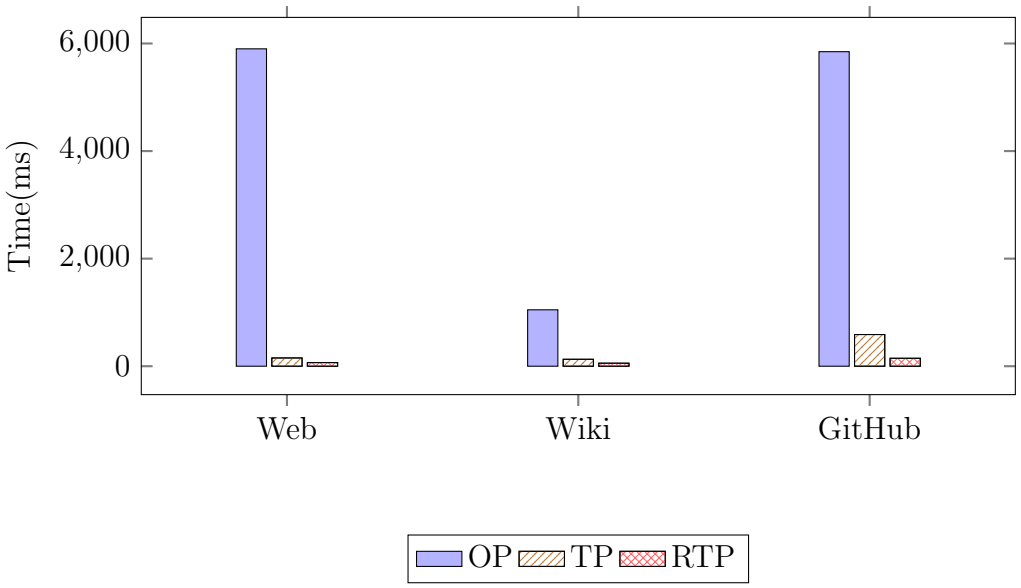


Figure 2.7: Query processing time of different search methods when the index is on SSD

of all document versions even positional information is skipped. Phase 1 of RTP has more time advantage for the GitHub dataset in which there are more versions per document. Note that TP's Phase 1 time reported here is higher than "a few ms" reported in [56]. This is because K parameter selected for Phase 1 in our experiment is larger in order to improve relevance.

Comparing TP and RTP's Phase 2 time, RPT is 5.41x as fast as TP on Web dataset, 4.16x on Wiki dataset and 3.86x on GitHub dataset. While our optimization plays a significant role, one reason is that extracting the positional information in [56] uses a global term-fragment index. Following our analysis on

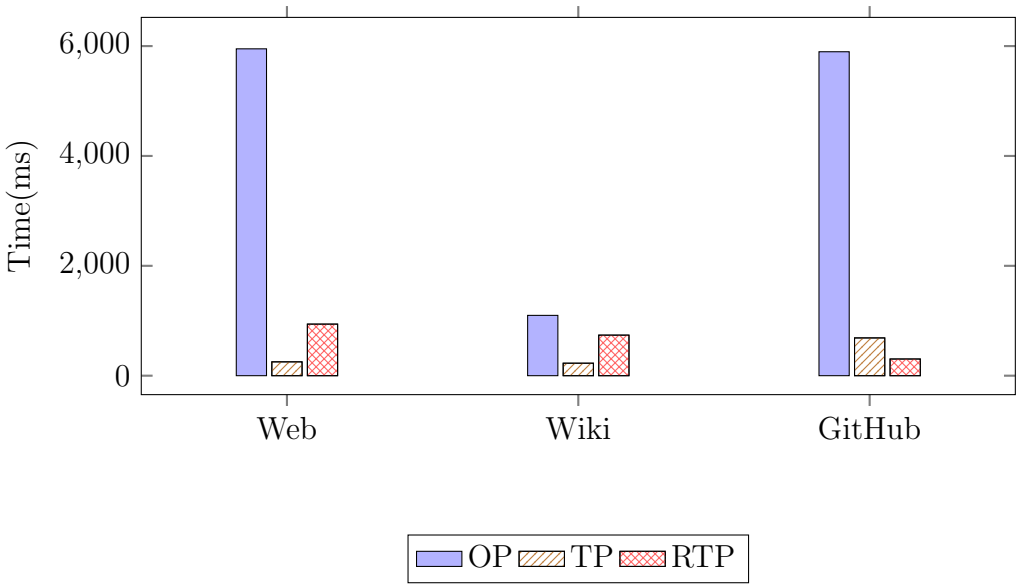


Figure 2.8: Query processing time of different search methods when the index is on HDD

Option B in Section 2.4.1, if we replace cost of searching local reuse table to a global reuse table, then the cost of position information extraction increases by a ratio of $\frac{\log F_i}{\log f_i}$ where F_i is the average posting length of global term-fragment index, which is much larger than f_i. This corroborates a benefit of cluster-based retrieval.

Table 2.3 shows the total processing time per query including the index loading overhead from an SSD or HDD. The gap between TP and RTP is narrowed because RTP has to access the index per version cluster separately and this results in more random disk block reads. Fortunately the fast SSD seek time for random I/O

still allows RTP to outperform others. RTP is 2.28x as fast as TP on the Web dataset, 2.30x on Wiki and 3.99x on GitHub. RTP is 87.7x as fast as OP on the Web dataset, 18.8x on Wiki and 39.7x on GitHub. The SSD I/O cost is about 44%, 41%, or 4% of the overall time, respectively for these datasets. Searching the GitHub dataset incurs the smallest I/O cost percentage because the number of versions per cluster is the highest and its k value is the smallest. The above I/O cost may be further reduced in the future with more code optimization. When the index resides in HDD, OP is still the slowest because its slow performance in Phase 1. RTP is 6.34x as fast as OP on the Web dataset, 1.49x on Wiki and 19.4x on GitHub. Comparing to TP, the time advantage of RTP diminishes in Phase 2 because of high random I/O cost. For GitHub with 439 versions per document, RTP is 2.26x as fast as TP. For Web wth 20 versions per page and Wiki with 13 versions per document, TP is 3.72x and 3.25x as fast as RTP. Thus we recommend RTP to be used when the search index can be stored in an SSD. It is suitable for HDDs only when handling a very large number of versions.

Time (ms)	Position extraction	Intersection	Scoring
Web	9.570	0.24	11.55
Wiki	15.87	0.25	12.69
GitHub	96.47	1.44	41.77

Table 2.4: Cost distribution of RTP at Phase 2

2.5.3 A Comparison of Phase 2 Indexing and Traversal Options

Table 2.4 shows the cost distribution of RTP in-memory search time at Phase 2. The feature extraction time is significant and this demonstrates the importance of reducing conversion time in Phase 2 computation.

Time (ms)	Web	Wiki	GitHub
Option A	35.74	20.81	128.5
Option B	9.778	23.57	124.6
Option C	9.57	15.87	96.47
Cluster-choice A	36.34%	91.67%	89.37%
Optimum	8.472	13.55	77.67

Table 2.5: In-memory search time with different options for Phase 2.

Table 2.5 lists the average query processing time of Phase 2 with various options. Row marked "Optimum" is computed by choosing the minimum value among Option A and B for each query in searching each cluster. The switch condition listed in Algorithm 2 may not find the best choice in all cases and thus this row represents the lower time bound an optimum algorithm may achieve. Row marked "Cluster-choice A" lists the percentage of the clusters that choose Option A as the traversal method decided by Option C. Option B can be 265% faster than Option A for Web, but can be 13% slower for Wiki. Option C adaptively predicts the winner between Option A and B, and is getting closer to what the optimum can accomplish. Compared to Option A and B, Option C is up-to 273% faster for Web , 48.5% for Wiki, and 33.2% for GitHub.

Why is Option A slower than Option B even 89.37% of clusters choose Option A traversal? The reason is that choosing Option A gives an improvement of 0.5523ms per cluster than choosing Option B in this case. In comparison, 10.63% clusters choose Option B, which gives an improvement of 5.484ms per cluster than choosing Option A. In general, we find that choosing Option A often delivers a relatively smaller time reduction while taking Option B often yields a bigger reduction. One reason is that Option B is often chosen when handling popular words, which tend to carry more shares of the overall search cost. For Wiki, choosing Option A reduces time by 0.0430ms per cluster while time reduction is

0.304ms per cluster when choosing Option B. For Web, time reduction 0.01191ms per cluster than choosing Option A and it is 0.8575ms per cluster when choosing Option B.

Index(MB)	Web	Wiki	GitHub
Option A	907	614	264
Option B	1225	905	452
Option C	1411	1008	654
Global	1237	1044	991

Table 2.6: Phase 2 index size of different options

Index(MB)	Web	Wiki	GitHub
Posting bitvectors	323	294	155
Term-fragment index	721	511	62
Fragment-version reuse table	186	103	202
Version-fragment mapping	181	100	235

Table 2.7: Phase 2 index size of different structures

Table 2.6 shows the compressed Phase 2 index storage size in megabytes under different options. Note that the uncompressed index size can be upto an order of magnitude larger than the numbers reported here. The row marked with "Global"

combines all data structures without cluster separation as explained at the end of Section 2.4.3. The order of the space usage is Option C, Option B and Option A. Option C does use more space for accomplishing the fastest processing time. For Web and Wiki datasets in which V is modest, the space difference ratio between Option A and Option B is about 26% and 32%. This is close to our storage cost analysis in Section 2.4.3 which estimates an approximated difference bound as 23%. From GitHub's result in which the number of version is large, the space difference ratio between Option B and Option C is about 31%. That is also within the estimated upper bound 50%.

Option C has similar cost as the global index for Wiki. For GitHub, the global index takes about 50% more space than Option C, which is about the same as what the space anlysis has estimated when V is very large. For Web with a modest number of versions, the global index uses 12.3% less space than Option C, which does not exceed the estimated upper bound 28.8%.

Table 2.7 shows the size of different components in our index. For Web and Wiki's results with a modest number of versions, term-to-fragment index is the largest component. On the other hand, for GitHub dataset, term-to-fragment index becomes less significant. That is because the fragment-based compression [118] becomes more effective for a large number of versions.

2.5.4 A Comparison on Relevance Scores

For single-word queries, the relevance of RTP is about the same as that of TP and OP and we report the results of relevance evaluation on queries with two or more words. We have randomly sampled 50 queries per dataset with a distribution following the log study in [8]. Here are some sample queries on the three datasets: 1) Web: dentist insurance, teaching assistant salary, student research grant application, international student center visa application; 2) Wiki: heart disease, England football team, second world war death, united nations security council members; 3) GitHub: Ethernet adapter, virtual memory access, virtual device data block, Linux kernel boot load device driver. There are four students involved in rating the final results of the different search methods with a score leveled from 0 to 3. Here relevance level 3 means the selected version is perfect for answering the query and level 0 means irrelevant. The names of search methods are not revealed to the evaluators as we union results from all algorithms together. Thus there is no bias in the rating process.

The default representative of RTP uses a superdocument that starts from the longest version (SLO) as the basis and then includes all words from versions. We have compared another way of selecting the representatives: the superdocument starts from the latest version(SLA).

vNDCG1@10		Web	Wiki	GitHub
OP		0.6478	0.7012	0.6889
K=200, 130, 4390	TP	0.4268	0.4833	0.5166
(k=10)	RTP	0.6157	0.6460	0.6137
K=400, 260, 8780	TP	0.5856	0.6364	0.6253
(k=20)	RTP	0.6557	0.6974	0.6664
K=1000, 650, 21950	TP	0.6460	0.6888	0.6780
(k=50)	RTP	0.6560	0.6988	0.6782
K=2000, 1300, 43900	TP	0.6468	0.6912	0.6823
(k=100)	RTP	0.6562	0.6988	0.6868

Table 2.8: Relevance scores of RTP compared with the other methods in terms of vNDCG@10 for different k values and K=k*V

Table 2.8 lists the vNDCG1@10 results of RTP with SLO method compared to OP and TP for the three datasets. The table lists the number of top documents (K) selected at Phase 1 for TP and the corresponding number of top clusters (k) selected at Phase 1 at RTP. For example, "K=200, 130, 4390 (k=10)" means that TP selects top 200, 130, and 4390 for three datasets Web, Wiki and GitHub

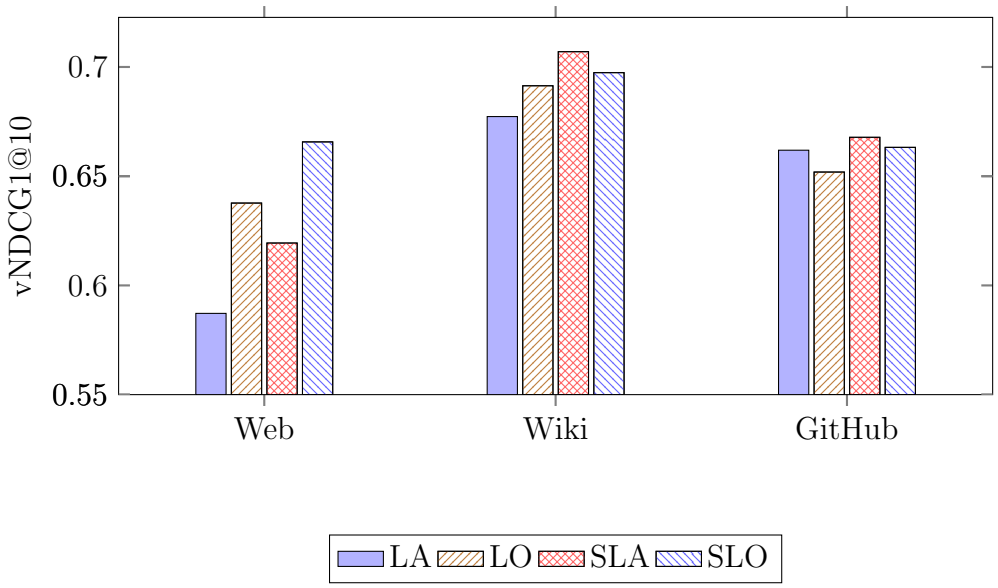


Figure 2.9: Impact of representative selection on vNDCG1@10 relevance scores

respectively while RTP selects top 10 clusters which contain K pages in total. For a smaller K (and k), RTP is doing better than TP by taking the proximity into account earlier. Still both RTP and TP have an insufficient coverage of relevant results in Phase 1 and thus have a relatively lower score compared to OP. When increasing the number of top results in Phase 1, the relevancy gap between TP and RTP becomes smaller and also both are getting closer to OP.

Figure 2.9 depicts the vNDCG1@10 scores of different representative selection options when k is 20 and SLO is the best choice among these options. The relevance score of SLO is 7.5% higher than SLA for Web, 1.3% lower for Wiki, and 0.7% lower for GitHub. On average, SLO is better than SLA by 1.83%. We

also assess the impact of adding all words from versions to the superdocuments. LA means using the latest version without adding words from other documents. LO means using the longest version without adding words from other documents. For all three datasets, the superdocument-based selection (SLA or SLO) is in general more effective than LA or LO. For the Wiki dataset, SLA is 4.39% better than LA and SLO is 0.87% better than LO. For the Web dataset, SLA is 5.48% better than LA and SLO is 4.39% better than LO. For GitHub data, SLA improves 0.89% over LA and SLO improves 1.73% over LO. The longest version is more effective in representing the positional information. Adding the extra terms in a super version provides 2.33% relevance improvement on average.

Ratio	LA	LO	SLA	SLO
Web	50.32%	95.19%	100.00%	100.00%
Wiki	94.26%	98.73%	100.00%	100.00%
GitHub	92.64%	96.60%	100.00%	100.00%

Table 2.9: Word coverage of the four representative selection methods

Table 2.9 shows the average distinct word coverage ratio (AWCR) for the four representative selection algorithms. Word coverage ratio of a document group is calculated by dividing the total number of distinct words in a document repre-

sentative by that of all versions of this document. AWCR is the average word coverage ratio of all documents. Since SLA and SLO have a non-positional index which covers words from all versions of a document, the AWCR value of SLA and SLO is 100%. LO has a word coverage 89% higher than LA for the Web dataset, which indicates SLO has a significantly better positional information coverage than SLA. This explains why vNDCG score of LO is 8.6% better than LA and the score of SLO is 7.5% better than SLA in Figure 2.9. For other datasets, the AWCR value of LA is closer to LO in Table 2.9 and that explains the relevance score of SLA and SLO is close in Figure 2.9.

2.6 Summary

In this chapter, we study the problem of, and propose a solution to, multiversion data search using positional index structures. In particular, the main contribution of this chapter is a hybrid indexing method with adaptive runtime traversal in supporting fast two-phase versioned data search and an integration with cluster-based retrieval using guided representatives. Our evaluation with a prototype implementation using three datasets shows the following results.

- RTP has a significant efficiency advantage on SSDs. It is suitable for HDDs only when there is a very large number of versions. RTP can be 3.83x to

4.12x as fast as the TP method if the search index is in memory. When the overhead of loading the index from an SSD is included, RTP can be 2.28x to 3.99x as fast as TP. Both approaches can be one or two orders of magnitude faster than a classical one-phase algorithm on SSDs while delivering competitive relevancy with a proper choice of top K or k value.

- The hybrid index with adaptive traversal (Option C) can be up-to 273% faster than Options A and B in Phase 2 in-memory query processing. Option C design represents a time-space tradeoff as Option A can use up-to 59.6% less compressed space while Option B can use up-to 30.9% less. On average, the space cost of Option C is more or less comparable to that of a global index.

The proposed work is focused on conjunctive queries and one future study is to consider disjunctive queries. Another future study is to investigate the incremental index update with time-based partitioning. When a new version is added, fragments shared with other document versions need to be identified and an approximation under a certain time interval may be applied for cost reduction. The Phase 1 index may be changed following the traditional index update techniques if the cluster representative changes and the update for a cluster index is fairly local.

Chapter 3

Cache-Conscious Runtime Optimization for Ranking Ensembles

3.1 Introduction

Learning ensembles based on multiple trees are effective for web search and other complex data applications (e.g. [49, 33, 51]). It is not unusual that algorithm designers use thousands of trees to reach better accuracy and the number of trees becomes even larger with the integration of bagging. For example, winning teams in the Yahoo! learning-to-rank challenge [33] have all used boosted

regression trees in one form or another and the total number of trees reported for scoring ranges from 3,000 to 20,000 [52, 24, 53], or even reaches 300,000 or more combined with bagging [87]. Generally speaking, application training data with less attributes may require smaller trees or a smaller number of trees. But as complex applications evolve over the time, more attributes are augmented and more trees can yield better accuracy.

Computing scores from a large number of trees is time-consuming. It is very expensive because each tree is traversed with branching conditions to compute an individual score for each document. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and instruction branch prediction. Compiler optimization [13] cannot handle complex code such as rank scoring very well. As a result, the query processing time or throughput is affected by using a large number of ensemble trees.

For example, processing a 150-leaf (at most 150 leaves per tree) 8,051-tree ensemble can take up to 3.04 milliseconds for a document with 519 features on an AMD 3.1 GHz core. Thus the scoring time per query exceeds 6 seconds to rank the top-2,000 results. It takes more time proportionally to score more documents with larger trees or more trees and this is too slow for interactive query performance. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel. That

is too slow for a real-time search system, even it is parallelizable with multiple cores. Today's search engines require a query processing time be completed within several hundreds of milliseconds when result caching is not used and a highly efficient execution of a tree-based ranking model is desirable. which stimulates the research of efficiently ranking algorithm even more.

Tradeoff between ranking accuracy and performance can be played by using earlier exit based on document-ordered traversal (DOT) or scorer-ordered traversal (SOT) [26], and by tree trimming [11]. The work in [12] proposes an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch mis-prediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient. One weakness is that the length of the enrolled code is quadratic to the depth of a tree and it is hard to maintain such code. For example, a tree with depth of 55 requires over 20,000 lines of unrolled C code.

In a modern CPU architecture, unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. How can fast multi-tree ensemble ranking with simple code structure be accomplished via memory hierarchy optimization, without compromising ranking accuracy? This is the focus of this chapter.

We propose a cache-conscious 2D blocking method to optimize data traversal for better temporal cache locality. Our experiments show that 2D blocking can be up to 620% faster than DOT, up to 245% faster than SOT, and up to 50% faster than VPred. After applying 2D blocking on top of VPred which shows advantage in reducing branch mis-prediction, the combined solution Block-VPred could be up to 100% faster than VPred. The proposed techniques are complementary to previous work and can be integrated with the tree trimming and early-exit approximation methods.

3.2 Background and Related Work

In this section, we will provide background and related work. Firstly, we will discuss learning to efficiently rank, which is how to do fast ranking on a given learned model. Here we mainly talk about tree-based model because it is the most state-of-the-art one. Secondly, we will discuss different traversal patterns especially the document-ordered traversal (DOT) and scorer-ordered traversal (SOT). We will also introduce VPred which is our major baseline method in this work. At last, we will talk about search engine caching technique briefly which is related but more orthogonal to our work here.

3.2.1 Learning to Efficiently Rank

Document ranking was treated by manually designing ranking functions in the past (e.g. BM25). Recently, supervised learning has been proved to be the most effective way to solve this problem. One of the ealist work among which is RankNet [23], which uses a neural network as a training model. Besides this, a lot of other famous methods are proposed by researchers, e.g. MART [48], RankBoost [47], AdaRank [112], Coordinate Ascent [78], LambdaMART [111], ListNet [28], Random Forests [20] and so on. To provide a fair comparison of all the ranking models, Yahoo! learning-to-rank challenge [33] invites researchers all over the world to participate in a contest in ranking. Boosted regression trees are proved to be the most effective learning models from this contest. In this chapter, we will discuss how to increase query throughput under the setting of the most effective ranking model: boosted regression trees.

We first define the problem of tree-based ensemble ranking. Given a query, there are n documents matching this query and the ensemble model contains m trees. Each tree is called a scorer and contributes a subscore to the overall score for a document. The final ranking score is a summation over all subscores.

Gradient Boosted Regression Tree algorithm which has been proved to be highly effective in learning to rank problems. However, the more accurate results

we want, the more complex model(more ensembles) we need. Therefore, efficiency and effectiveness is a tradeoff in the ranking problem.

There are a number of performance speedup techniques proposed in the previous work to speedup fast ranking score computation, which can be summarized into two categories.

- The first category is to achieve a tradeoff between ranking efficiency and accuracy. In the early time, researchers focus more on achieving a efficiency-effectiveness tradeoff. In [27], Noticing that most users pay more attention to the first few pages of the ranking results and document relevance follows a skewed distribution, an early exit optimization was developed to reduce scoring time while retaining a good ranking accuracy. In [108, 109], ranking is optimized to seek the tradeoff between efficiency and effectiveness. In [109], Wang et. al proposed a cascade ranking model that progressively prunes and refines the candidate documents set to minimize retrieval latency and maximize accuracy. Asadi et.al [11] considered the fact that compact, shallow, and balanced trees yield faster computation and generated such trees with trimming technique.

- The second category is to improve efficiency given a fixed model. In recent years, there has been several research work on improving ranking efficiency without affecting accuracy. The work in [12] proposed an architecture-

conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch misprediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient.

Our method belongs to the second category. We propose a 2D framework which can better use cache locality to improve ranking efficiency while does not affect accuracy at all. Also, noting that our framework does not only apply to tree-based ensemble ranking, it also works with many other models and is a generic one.

3.2.2 Traversal Patterns

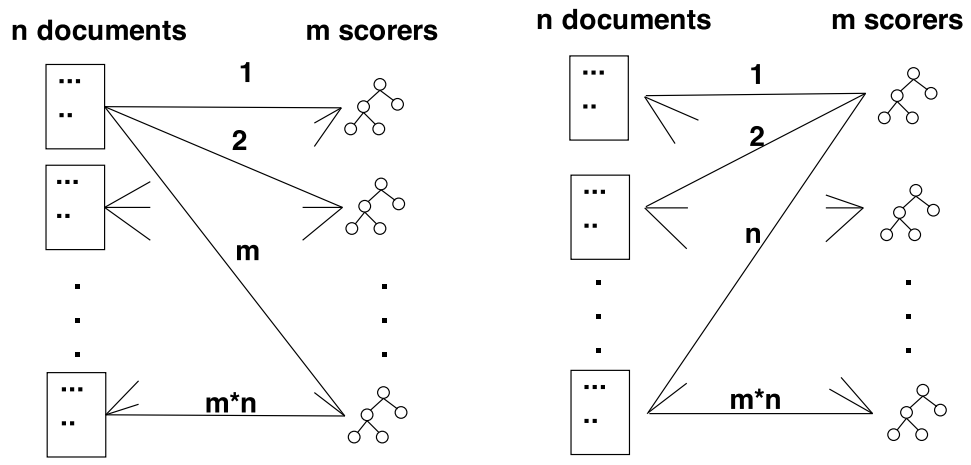
Following the notation in [26], Algorithm 3 shows the program of DOT. At each loop iteration i, all trees are calculated to gather subscores for a document before moving to another document. In implementation, each document is represented as a feature vector and each tree can be stored in a compact array-based format [12]. The time and space cost of updating the overall score with a subscore is relatively insignificant. The dominating cost is slow memory accesses during tree traversal based on document feature values. By exchanging loops i and j in Algorithm 3, DOT becomes SOT. Their key difference is the traversal order. In

SOT a tree is used to compute subscores for all documents first before visiting next tree. Operations on test instances are interleaved, and one scorer is applied to all instances before moving to the next. The outer loop iterates on scorers, so it requires keeping all feature vectors in memory until the last scorer is executed. SOT and DOT exhibit different performance behavor. Previous research [26] favored DOT due to its good cache hit rates in accessing feature vectors, as well as the convenience to de-allocate an instance vector once it is scored.

```
1 for i = 1 to n do
2     for j = 1 to m do
3         Compute a subscore for document i with tree j.
4         Update document score with the above subscore.
5     end
6 end
```

Algorithm 3: Ranking score calculation with DOT.

Figure 3.1(a) shows the data access sequence in DOT, marked on edges between documents and tree-based scorers. These edges represent data interaction during ranking score calculation. DOT first accesses a document and the first tree (marked as Step 1); it then visits the same document and the second tree. All m trees are traversed before accessing the next document. As m becomes large,



(a) (b)

Figure 3.1: Data access order in DOT (a) and SOT (b).

the capacity constraint of CPU cache such as L1, L2, or even L3 does not allow all m trees to be kept in the cache before the next document is accessed. The temporal locality of a document is exploited in DOT since the cached copy can be re-accessed many times before being flushed; however, there is no or minimal temporal locality exploited for trees. Similarly, Figure 3.1(b) marks data interaction edges and their access order in SOT. SOT traverses all documents for a tree before accessing the next tree. Temporal locality of a tree is exploited in SOT; however, there is no or minimal temporal locality exploited for documents when n is large.

VPred [12] converts if-then-else branches to dynamic data accesses by unrolling the tree depth loop. The execution still follows DOT order, but it overlaps the score computation of several documents to mask memory latency. Such vector-

ization technique also increases the chance of these documents staying in a cache when processing the next tree. However, it has not fully exploited cache capacity for better temporal locality. Another weakness is that the length of the unrolled code is quadratic to the maximum tree depth in a ensemble, and linear to the vectorization degree v. For example, the header file with maximum tree depth 51 and vectorization degree 16 requires 22,651 lines of code. Long code causes inconvenience in debugging and code extension. In comparison, our 2D blocking code has a header file of 159 lines.

Our block-based algorithm aims at better utilizing cache to reduce cache miss rate and reduce the runtime cost. While the processor is waiting for memory access for one instance, useful computation can happen on another. Rooted from DOT and SOT, our algorithm does not take the extreme of either, but targeting the block-based traversal pattern and optimal block setting to increase speed.

3.2.3 Search Engine Caching Techniques

Another thread of research which is orthogonal to our work is search engine caching techniques. Over the years, many caching techniques have been investigated and used in search engines [14, 15, 16, 44, 77, 84, 85, 86]. In recent years, research focus transfers to cache design on SSD disks [107, 103]. Search engine cache is indeed helpful to improve query throughput. However, our work is or-

thogonal to them so our research can be added to a search system on top of any of these caching techniques.

3.3 2D Block Algorithm

Algorithm 4 is a 2D blocking approach that partitions the program in Algorithm 3 into four nested loops. The loop structure is named SDSD because the first (outer-most) and third levels iterate on tree-based Scorers while the second and fourth levels iterate on Documents. The inner two loops process d documents with s trees to compute subscores of these documents. We choose d and s values so that these d documents and s trees can be placed in the fast cache under its capacity constraint. To simplify the presentation of this part, we assume $\frac{m}{s}$ and $\frac{n}{d}$ are integers. The hierarchical data access pattern is illustrated in Figure 3.2. The edges in the left portion of this figure represent the interaction among blocks of documents and blocks of trees with access sequence marked on edges. For each block-level edge, we demonstrate the data interaction inside blocks in the right portion of this figure. Note that there are other variations of 2D blocking structures: SDDS, DSDS and DSSD. Our evaluation finds that SDSD is the fastest for the tested benchmarks.

There are two to three levels of cache in modern AMD or Intel CPUs. For the tested datasets, L1 cache is typically too small to fit multiple trees and multiple

```
1  Initialize scores to be zero;
2  for j = 0 to m/s − 1 do
3      for i = 0 to n/d − 1 do
4          for jj = 1 to s do
5              for ii = 1 to d do
6                  Compute subscore for document i × d + ii with tree j × s + jj.
7                  Update the score of this document.
8              end
9          end
10     end
11 end
```

Algorithm 4: 2D blocking with SDSD structure.

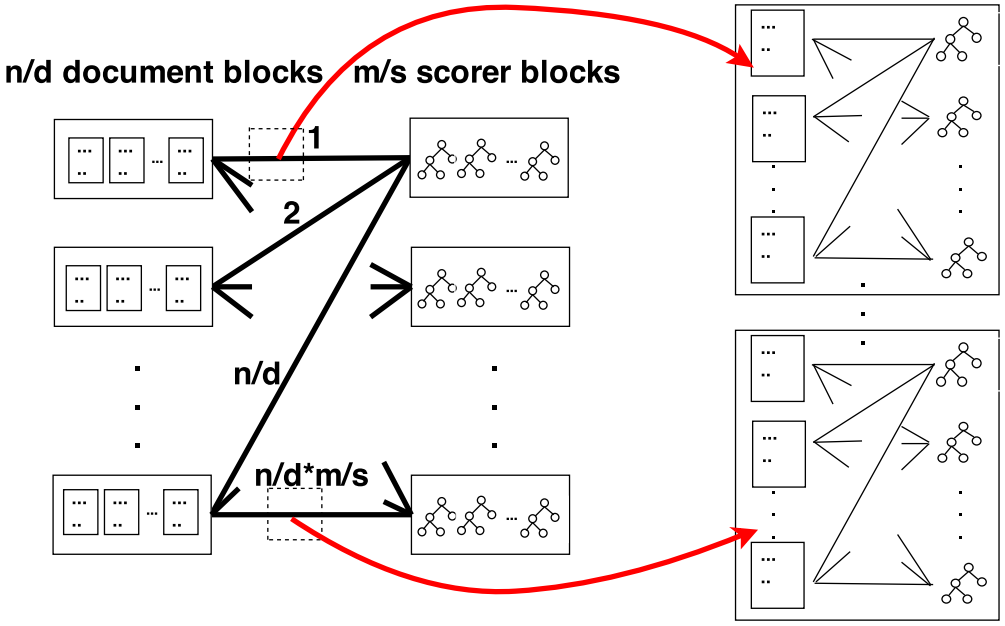


Figure 3.2: Data access order in the SDSD blocking scheme.

document vectors for exploiting temporal locality. Thus L1 is used naturally for spatial locality and more attention is on L2 and L3 cache. 2D blocking design allows the selection of s and d values so that s trees and d documents fit in L2 cache.

Detailed cache performance analysis requires a study of cache miss ratio estimation in multiple levels of cache. We use a simplified cache-memory model to illustrate the benefits of the 2D blocking scheme. This model assumes there is one level of cache which can hold d document vectors and s tree-based scorers, i.e. space usage for s and d do not exceed cache capacity. Here we estimate the total slow memory accesses during score calculation using the *big O* notation. The inner-most loop ii in Algorithm 4 loads 1 tree and d document vectors. Then loop jj loads another tree and still accesses the same d document vectors. Thus there are a total of $O(s) + O(d)$ slow memory accesses for loops jj and ii. In loop level i, the s trees stay in the cache and every document block causes slow memory accesses, so memory access overhead is $O(s) + O(d) \times \frac{n}{d}$. Now looking at the the outer-most loop j, total memory access overhead per query is $\frac{m}{s}(O(s) + O(n))$ $= O(m + \frac{m \times n}{s})$.

From Figure 3.1, memory access overhead per query in DOT can be estimated as $O(m \times n + n)$ while it is $O(m \times n + m)$ for SOT. Since term $m \times n$ typically

dominates, our 2D blocking algorithm incurs s times less overhead in loading data from slow memory to cache when compared with DOT or SOT.

Vectorization in VPred can be viewed as blocking a number of documents and the authors have reported [12] that a larger vectorization degree does not improve latency masking and for Yahoo! dataset, 16 or more degree performs about the same. The objective of 2D blocking scheme is to fully exploit cache locality. We can apply 2D blocking on top of VPred to exploit more cache locality while inheriting the advantages of VPred. We call this approach Block-VPred. The code length of Block-VPred is about the same as VPred.

In order to estimate the access cost of block-based approach, we break down the cost into accessing different level of caches or memory. The efficiency of algorithms could be evaluated using a cost ratio

$$\delta_1 + m_1(\delta_2 - \delta_1) + m_1 m_2(\delta_3 - \delta_2) + m_1 m_2 m_3(\delta_{mem} - \delta_3). \tag{3.1}$$

Instead of processing all the documents or all the scorers at-a-time, we break them each into several blocks, and process each block pair at a time. By varying the number of documents processed at a time (d) and the number of scorers processed at a time (s), we fit partial forward index and partial ensemble trees in cache, achieving high cache utilization. As shown in Algorithm 4, block-based

Case	m_1	m_2	m_3	m_{mem}	Description
(D1)	1	$1/s$	0	0	D fits L2.
(D2)	1	1	$1/s$	0	D fits L3.
(D3)	1	1	1	$1/s$	D fits memory.
(S1)	$1/d$	d/n	0	0	S fits L2.
(S2)	$1/d$	1	d/n	0	S fits L3.
(S3)	$1/d$	1	1	d/n	S fits memory.

Table 3.1: Cases of cache miss ratios for area S and D when fit different levels of cache.

approach loop through m/s scorer blocks and n/d document blocks, where each block evaluate d document against s scorers.

For each block, we illustrate the memory allocation in In tree-based models like GBRT, each element is a regression tree. Within a tree, each node stores a feature (f_i) and a threshold (t_i) to decide whether to proceed to left or right subtree. A regression value is returned at some leaf node. The document is stored in the format of forward index where arrays of weights (w_j) are stored adjacently. For a tree-based scorer with average depth L, a condition in a tree and a corresponding weight in a document are accessed for each step, and the step is repeated L times before a score is reached. Hence the total number of

data load is $D_0 = 2mnL$ considering both tree and documents. We estimate the cache miss ratios for accessing data in D and S respectively, and list several cases in Table 3.1. For example, if D fits L2 cache (case D1), the number of L2 cache miss m_2 could be reduced to $1/s$ because for s scorers inside a block, d document weights are fetched to L2 cache only once. The situation for area S is slightly different because scorers are the outer loop of the block computation. Each time a scorer vector is fetched, the next $d-1$ accesses are benefited because we do not need to be fetched again for the d documents in the instance block. This explains $m_1 = 1/d$ for cases S1, S2 and S3. If S fits L2 cache (case S1), it stays in cache when comparing with documents from the first instance block till the n/d_q block, so $m_2 = d/n$ in this case.

We illustrate s and d values for the optimal case as follows. Once we feed the ratios in Table 3.1 to Equation 3.1, we discover that most cache misses happen when accessing area D since $m_1(D) \ll m_1(S)$. The optimal choice occurs in the upper boundary of case D1, when the documents in instance block fits right in L2. It is clear that cases D2 and D3 are worse than D1 due to higher m_3 or m_{mem} values.

3.4 Evaluations

3.4.1 Datasets and Experiment Settings

Settings. 2D block and Block-VPred methods are implemented in C and VPred code is from [12]. Code is compiled with GCC using optimization flag -O3. Memory is allocated for a whole tree with the maximum number of leaf nodes at a time. The whole ensemble is stored adjacently in memory as an array of trees. Experiments are conducted on a Linux server with 8 cores of 3.1GHz AMD Bulldozer FX8120 and 16GB memory. FX8120 has 16KB of L1 data cache per core, 2MB of L2 cache shared by two cores, 8MB of L3 cache shared by eight cores. The cache line is of size 64 bytes. Experiments are also conducted in Intel X5650 2.66GHz six-core dual processors and the conclusions are similar. In this dissertation, we report the results from AMD processors.

Datasets. We use the following learning-to-rank datasets as the core test benchmarks.

- Yahoo! dataset [33] with 709,877 documents and 519 features per document from its learning-to-rank challenge.
- MSLR-30K dataset [2] with 3,771,125 documents and 136 features per document.

- MQ2007 dataset [1] with 69,623 documents and 46 features per document.

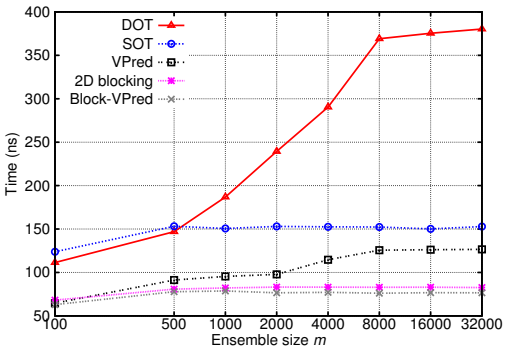
The tree ensembles are derived by the open-source jforests [51] package using LambdaMART [24]. To assess score computation in presence of a large number of trees, we have also used bagging methods to combine multiple ensembles and each ensemble contains additive boosting trees.

There are 23 to 120 documents per query labeled in these datasets. In practice, a search system with a large dataset ranks thousands or tens of thousands of top results after the preliminary selection. We synthetically generate more matched document vectors for each query. Among these synthetic vectors, we generate more vectors bear similarity to those with low labeled relevance scores, because typically the majority of matched results are less relevant.

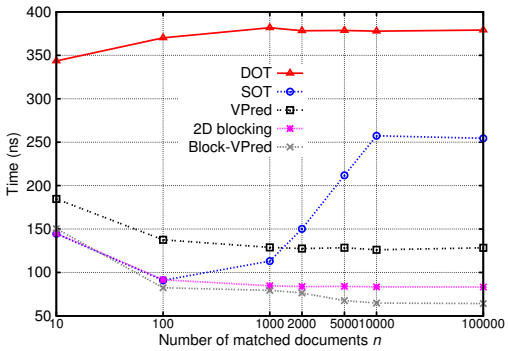
Metrics. We mainly report the average time of computing a subscore for each matched document under one tree. This scoring time multiplied by n and m is the scoring latency per query for n matched documents ranked with an m-tree model. Each query is executed by a single core.

3.4.2 A Comparison of Scoring Time

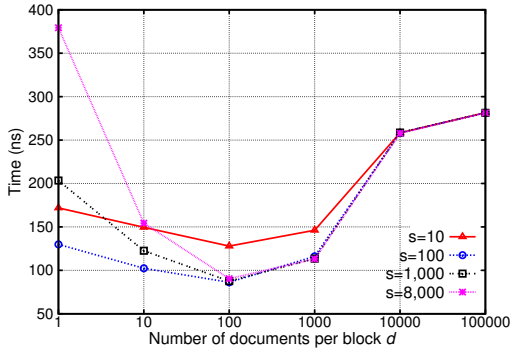
Table 3.2 lists scoring time under different settings. Column 2 is the maximum number of leaves per tree. Tuple [s,d,v] includes the parameters of 2D blocking and the vectorization degree of VPred that leads to the fastest scoring time. Choices of



(a)



(b)



(c)

Figure 3.3: Scoring time per document per tree in nanoseconds when varying m (a) and n (b) for five algorithms, and varying s and d for 2D blocking (c). Benchmark used is Yahoo! dataset with a 150-leaf multi-tree ensemble.

Dataset	Leaves	m	n	DOT	SOT	VPred [v]	2D blocking [s, d]	Block-VPred [s, d, v]	Latency
Yahoo!	50	7,870	5,000	186.0	113.8	47.4 [8]	36.4 [300, 300]	36.7 [300, 320, 8]	1.43
	150	8,051	2,000	377.8	150.2	123.0 [8]	81.9 [100, 400]	76.1 [100, 480, 8]	1.23
	400	2,898	5,000	312.3	223.8	136.2 [8]	90.9 [100, 400]	86.0 [100, 400, 8]	1.25
MSLR-30K	50	1,647	5,000	88.3	41.4	32.6 [8]	26.6 [500, 1,000]	31.1 [500, 1,600, 8]	0.22
MQ2007	50	9,870	10,000	1.79	1.66	2.02 [8]	1.51 [300, 5,000]	1.94 [300, 5,000, 8]	0.15
	200	10,103	10,000	204.1	30.3	43.1 [32]	28.3 [100, 10,000]	26.2 [100, 5,000, 32]	2.65

Table 3.2: Scoring time per document per tree in nanoseconds for five algorithms. Last column shows the average scoring latency per query in seconds under the fastest algorithm marked in gray.

v for VPred are the best in the tested AMD architecture and are slightly different from the values reported in [12] with Intel processors. Last column is the average scoring latency per query in seconds after visiting all trees. For example, 2D blocking is 361% faster than DOT and is 50% faster than VPred for Row 3 with Yahoo! 150-leaf 8,051-tree benchmark. In this case, Block-VPred is 62% faster than VPred and each query takes 1.23 seconds to complete scoring with Block-VPred. For a smaller tree in Row 5 (MSLR-30K), Block-VPred is 17% slower than regular 2D blocking. In such cases, the benefit of converting control dependence as data dependence does not outweigh the overhead introduced.

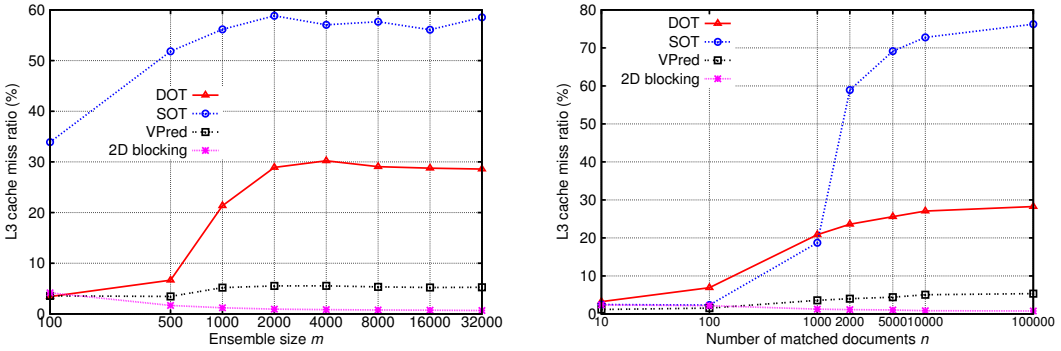
Figure 3.3 shows the scoring time for Yahoo! dataset under different settings. In Figure 3.3(a), n is fixed as 2,000; DOT time rises dramatically when m in-

creases because these trees do not fit in cache; SOT time keeps relatively flat as m increases. In Figure 3.3(b), m is fixed as 8,051 while n varies from 10 to 100,000. SOT time rises as n grows and 2D blocking is up to 245% faster. DOT time is relatively stable. 2D blocking time and its gap to VPred are barely affected by the change of m or n. Block-VPred is 90% faster than VPred when n=5,000, and 100% faster when n=100,000. Figure 3.3(c) shows the 2D blocking time when varying s and d. The lowest value is achieved with s=1,000 and d=100 when these trees and documents fit in L2 cache.

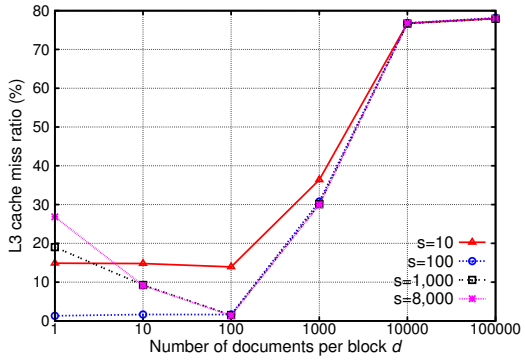
3.4.3 Cache Behavior

Linux *perf* tool reports L1 and L3 cache miss ratios during execution. We observed no strong correlation between L1 miss ratio and scoring time. L1 cache allows program to exploit limited spatial locality, but is too small to exploit temporal locality in our problem context. L3 miss ratio does show a strong correlation with scoring time. In our design, 2D blocking sizes (s and d) are determined based on L2 cache size. Since L2 cache is about the same size as L3 per core in the tested AMD machine, reported L3 miss ratio reflects the characteristics of L2 miss ratio.

Figure 3.4 plots the L3 miss ratio under the same settings as Figure 3.3 for Yahoo! data. This ratio denotes among all the references to L3 cache, how many are missed and need to be fetched from memory. The ratios of Block-VPred,



(a) (b)



(c)

Figure 3.4: L3 miss ratio when varying n (a), varying m (b) for four algorithms, and when varying s and d for 2D blocking (c).

which are not listed, are very close to that of 2D blocking. In Figure 3.4(a) with n=2,000, SOT has a visibly higher miss ratio because it needs to bring back most of the documents from memory to L3 cache every time it evaluates them against a scorer; n is too big to fit all documents in cache. The miss ratio of DOT is low when all trees can be kept in L2 and L3 cache; this ratio grows dramatically after m=500. Figure 3.4(b) shows miss ratios when m=8,051 and n varies. The miss ratio of SOT is close to VPred and 2D blocking when n¡100, but deteriorates significantly when n increases and these documents cannot fit in cache any more. The miss ratios of VPred in both Figure 3.4(a) and 3.4(b) are below 6% because vectorization improves cache hit ratio. Performance of 2D blocking is the best, maintaining miss ratio around 1% even when m or n is large.

Figure 3.4(c) plots L3 miss ratio of 2D blocking when varying s and d block sizes. The trends are strongly correlated with the scoring time curve in Figure 3.3(c). The optimal point is reached with s=1,000 and d=100 when these trees and documents fit in L2 cache. When s=1,000, miss ratio varies from 1.64% (d=100) to 78.1% (d=100,000). As a result, scoring time increases from 86.2ns to 281.5ns.

3.4.4 Branch Mis-prediction

We have also collected instruction branch mis-prediction ratios during computation. For MQ2007 and 50-leaf trees, mis-prediction ratios of DOT, SOT, VPred, 2D blocking and Block-VPred are 1.9%, 3.0%, 1.1%, 2.9%, and 0.9% respectively. For 200-leaf trees, these ratios increase to 6.5%, 4.2%, 1.2%, 9.0%, and 1.1%. VPred's mis-prediction ratio is lower than 2D blocking while its scoring time is still longer, indicating the impact of cache locality on scoring time is bigger than branch mis-prediction. For smaller trees, mis-prediction ratios of 2D blocking and Block-VPred are close and this explains why Block-VPred does not outperform 2D blocking in Table 3.2 for 50-leaf trees. Adopting VPred's strategy of converting if-then-else instructions pays off for large trees. For such cases when n increases, Block-VPred outperforms 2D blocking with lower branch mis-prediction ratios. This is reflected in the Yahoo! 150-leaf 8,051-tree benchmark: mis-prediction ratios are 1.9%, 2.7%, 4.3%, and 6.1% for 2D blocking, 1.1%, 0.9%, 0.84%, and 0.44% for Block-VPred, corresponding to the cases of n=1,000, 5,000, 10,000 and 100,000 respectively.

3.5 Summary

The main contribution of this work is cache-conscious design for computing ranking scores with a large number of trees and/or documents by exploiting memory hierarchy capacity for better temporal locality. Multi-tree score calculation of each query can be conducted in parallel on multiple cores to further reduce latency. Our experiments show that 2D blocking still maintains its advantage using multiple threads. In some applications, the number of top results (n) for each query is inherently small and can be much smaller than the optimal block size (d). In such cases, multiple queries could be combined and processed together to fully exploit cache capacity. Our experiments with Yahoo! dataset and 150-leaf 8,051-tree ensemble shows that combined processing could reduce scoring time per query by 12.0% when n=100, and by 48.7% when n=10.

Our 2D blocking technique is studied in the context of tree-based ranking ensembles and one of future work is to extend it for other types of ensembles by iteratively selecting a fixed number of the base rank models that can fit in the fast cache.

Chapter 4

A Comparison of Cache Blocking Methods

4.1 Introduction

Ensemble-based machine learning techniques have been proven to be effective for dealing data-intensive applications with complex features and document ranking is a representative application benefiting from use of the large number of ensembles. For example, in the Yahoo! learning-to-rank challenge [33], all winners have used some forms of gradient boosted regression trees, e.g. [48]. The total number of trees reported for ranking can be upto 3,000 to 20,000 [52, 24, 53], or even 300,000 or more using bagging method [87]. Ranking for large ensembles is

expensive. As reported in [99], it takes more than 6 seconds to rank the top-2000 results for a query processing a 8,051-tree ensemble and 519 features per document on an AMD 3.1 GHz core. If such an algorithm is used to compute scores for a large number of vectors in applications such as classification, the total job is also very time consuming. It takes even more time for a larger ensemble or for more candidate documents. The ranking process can be parallelized and the time can be reduced. However, it does not help for improving query throughput because less queries are processed in parallel.

The previous work addressed the speedup of runtime execution for ensemble-based ranking in several aspects including tree trimming [11] for a tradeoff of ranking accuracy and performance, earlier exit [27], and loop unrolling [12], and ensemble restructuring for a tree-based model [76]. Memory access can be 100x slower than L1 cache and un-orchestrated slow memory access incurs significant cost, dominating the entire computation. The work shown in [99, 76] proposes a cache-conscious blocking method for better cache locality. However, there are other block methods to select and it is an open problem how to identify the best cache blocking method and parameter settings given different data and architecture characteristics. Experimentally determining this choice can be extremely time-consuming and the comparative result may not be valid any more with a change of underlying feature vector structure or architecture. This chapter pro-

vides an analysis of multiple blocking methods with different data traversal orders, which provides better insight on program execution performance and leads a fast approximation to select the optimized structure for different application and architecture scenarios.

Here, we consider the fast computation of ensemble-based scoring that aggregates and derives final scores for n feature vectors using m ensembles. In addition to comparing the order of traversal in different cache blocking methods, we need to select d out of n feature vectors and s out of m scorers to form the innermost loop computation. For testing and comparing performance in ranking q sampled queries, the time cost for searching through all combinations can be as high as $O(m^2 * n^2 * q)$. What is a guideline to select a good combination of d and s? We will discuss this in this chapter.

The main contribution of this work is to develop an analytic framework to compare memory access performance of data traversal under multi-level caches to find the fastest program execution with effective use of memory hierarchy. Our scheme results in a much smaller complexity with $O(m * n * q)$ to assess and compare performance with q test queries. Our experiments with three datasets corroborate the effectiveness of search cost reduction while the guided approximation identifies a highly competitive blocking choice. We also demonstrate the

use of this scheme with QuickScorer [76] and for batched query processing that significantly accelerate the score calculation without loss of ranking accuracy.

The rest of the chapter is organized as follow. Next, we describe the background information and related work. Section 4.3 discusses the design considerations. Section 4.4 gives a comparative analysis on different blocking methods. Section 4.5 presents evaluation results. Finally, Section 4.6 concludes the chapter.

4.2 Background and Related Work

Given n feature vectors and an ensemble model that contains m scorers, these vectors and scorers fit in memory. The ensemble computation calculates a score for each feature vector and each scorer contributes a subscore to the overall score for a vector. For example, for ranking a document set with an additive regression tree model [48, 24], each document is represented as a feature vector and each tree can be stored in a compact array-based format [12]. Following the notation in [27], Algorithm 5 shows the DS method with the two-loop standard execution order. At each out loop iteration i, all scorers are used to gather subscores for a vector before moving to another vector. The dominating cost is slow memory accesses when scorers read feature vector values and update partial values.

Tang et. al [99] proposed a 2D cache blocking structure called SDSD as depicted in Algorithm 6 which partitions the program in Algorithm 5 into four

```
1 for i = 1 to n do
2 |   for j = 1 to m do
3 |   |   Update score for vector i with scorer j.
4 |   end
5 end
```

Algorithm 5: DS standard method for score calculation.

nested loops. The inner two loops process d feature vectors with s trees. To simplify the presentation, we assume n/d and m/s are integers. By fitting the inner block in fast cache, this method can be much faster than DS. There are other possible cache blocking methods with different data traversal orders and it is an unanswered question on how to choose among them. Also, in [99] there is no cost analysis on how to set a proper parameter for the size of blocking in terms of s and d values. While choices of their values can be restricted to fit in the fast cache, they can still be fairly large. For example, s and d can still reach upto 3,276 and 11,440 respectively in some of our experiments shown in Section 4.5. Assume m and n are smaller than these upper bound numbers, s ranges from 1 to m and d ranges from 1 to n and there are $m * n$ combinations to compare as they all fit in different levels of cache. Since running each test query takes $O(n*m*q)$, the total cost is $O(m^2 * n^2 * q)$. For instance, given $n = 10,000$, $m = 3,000$, $q = 1000$, the

total time takes over 1,141 years with one core, assuming it takes 40 nanoseconds to compute a partial score for a vector with a scorer. If we sample each of s and d values with step gap 100, the total one-core time is over 41 days without knowing if such sampling finds a solution competitive to the optimum. While running such a sampling can be fully parallelized, we still need a faster scheme with well-guided approximation.

```
1 for j = 0 to m/s − 1 do
2     for i = 0 to n/d − 1 do
3         for jj = 1 to s do
4             for ii = 1 to d do
5                 Update score for vector i × d + ii with scorer j × s + jj.
6             end
7         end
8     end
9 end
```

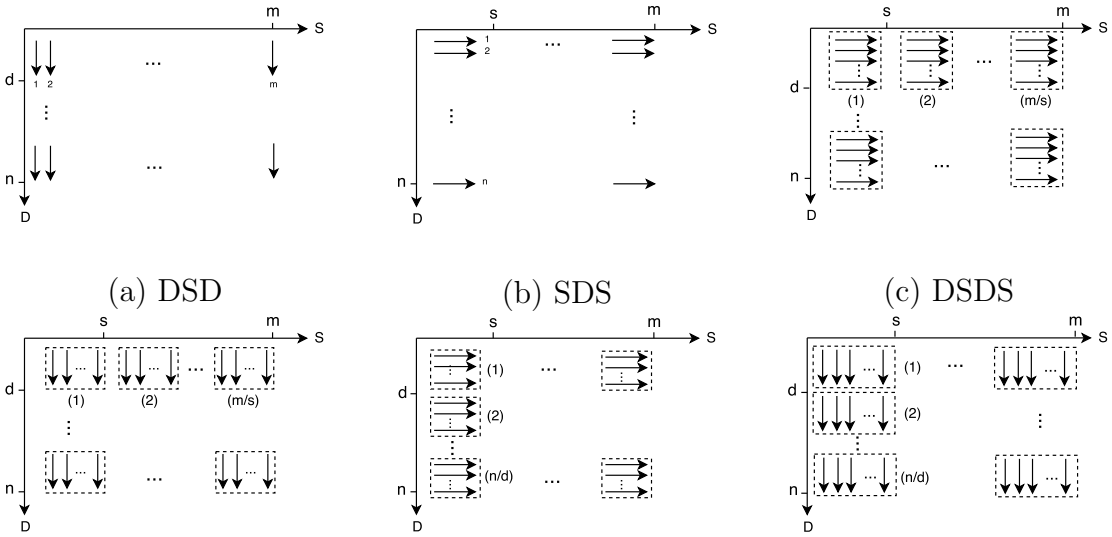
Algorithm 6: 2D blocking with SDSD structure.

There are other performance speedup techniques proposed in the previous work to speedup fast ranking score computation, which can be summarized into two categories. The first category is to achieve a tradeoff between ranking effi-

ciency and accuracy. In [27], an early exit optimization was developed to reduce scoring time while retaining a good ranking accuracy. In [108, 109], ranking is optimized to seek the tradeoff between efficiency and effectiveness. Asadi et.al [11] considered the fact that compact, shallow, and balanced trees yield faster computation and generated such trees with trimming technique. The second category is to improve efficiency given a fixed model. The work in [12] proposed an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization. Lucchese et.al proposed the QuickScorer (QS) algorithm [76] which traverses multiple trees in an interleaved manner and accelerates with bit-wise operations. They propose a block-wise variant of QS (called BWQS) by partitioning trees into blocks and applying QS to each block of trees. Given different dataset characteristics, it is an open problem how to find the optimal partitioning. Also there are other ways to arrange blocking and our work is complementary and can be used to compare different options.

4.3 Design Consideration and Cost Model

There are six ways of loop blocking depending on the order of data traversal: DSD, SDS, DSDS, DSSD, SDDS, and SDSD. Following the naming in [103], symbol D here stands for a loop control over feature vectors and S stands for a loop



(a) DSD (b) SDS (c) DSDS

(d) DSSD (e) SDDS (f) SDSD

Figure 4.1: Data traversal order of cache blocking methods during execution

control over scorers. For example, DSDS means that feature vector traversal is controlled by the outermost and the third outermost loops while scorer traversal is controlled by the second and the innermost loops. The inner two loops access d vectors and s scorers.

Figure 4.1 illustrates the execution and data traversal order of these methods. Figure 4.1(a) shows that DSD initially visits one scorer and d vectors. Then it visits another scorer and the same d vectors. Figure 4.1(b) depicts that SDS initially visits one vector and s scorers. Then it visits another vector and the same s scorers. Figure 4.1(c) illustrates DSDS which visits scorers and vectors

block by block and row by row. Figure 4.1(f) illustrates SDSD which visits scorers and vectors block by block and column by column.

Our objective is to compare these blocking methods and find a value for s and d to minimize the time cost of score computation under a constraint $1 \leq s \leq m,\ 1 \leq d \leq n$. We have the following considerations.

- For DSD, when the inner most loop uses $d = 1$, it becomes a special case DS which is the same as the traditional loop structure DS shown in Algorithm 5. For SDS, when the inner loop uses $s = 1$, it becomes a special case SD.

- The traversal order of DSSD during execution is the same as that of DSD as illustrated in Figure 4.1(a) and (d). Thus DSD can represent both during our analysis. Similarly, the traversal order of SDDS is the same as that of SDS as shown in Figure 4.1(b) and (e). As a result of the above argument, the six types of control are reduced to four.

The following parameters are used in assessing the average memory access cost of processing n feature vectors with m scorers. We assume that CPU has three levels of caches: L1, L2, and L3 and the three level setting is popular in the currently available processors from Intel and AMD. Let δ_1 be the read or write cost of accessing L1 and cost for accessing other cache is δ_1 multiplied by a constant

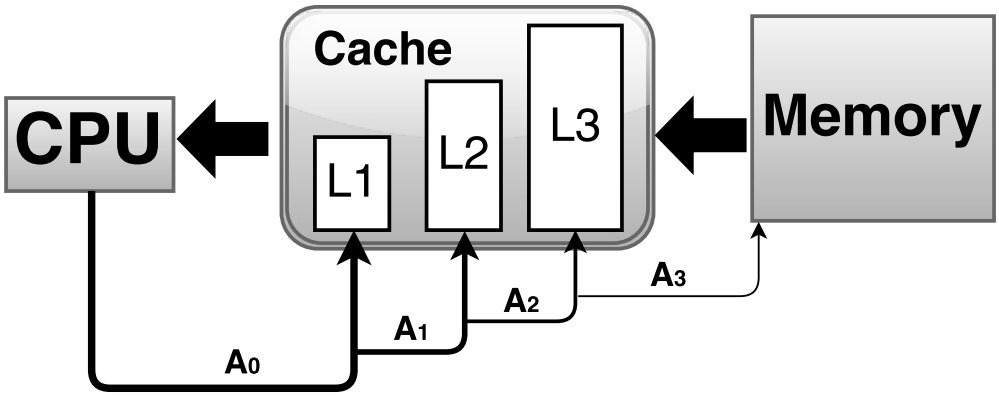


Figure 4.2: Data access flow of CPU with memory hierarchy.

ratio. Namely $c_2\delta_1$ is the cost of accessing L2, $c_3\delta_1$ is the cost of accessing L3, and $c_4\delta_1$ is the cost of accessing memory.

Our analysis separates the cost for accessing feature vectors and scorers. Without losing the generality, let A_i be the total amount of data access to feature vectors at cache level $i+1$ while ηA_i be the total amount of accesses to scorers at cache level $i+1$ and η is the average frequency ratio between access of feature vectors and scorers during computation.

The total data access cost is the summation of the cost of accessing each level of memory hierarchy:

$$Cost = A_0\delta_1(1 + \alpha_D c_2 + \alpha_D\beta_D c_3 + \alpha_D\beta_D\gamma_D c_4)$$

$$+ \eta(A_0\delta_1(1 + \alpha_S c_2 + \alpha_S\beta_S c_3 + \alpha_S\beta_S\gamma_S c_4)).$$

where α_S, β_S, γ_S, α_D, β_D, and γ_D are the miss rates of L1, L2 and L3 to access scorers and feature vectors respectively. Data accesses flow from CPU to memory

for feature vectors is illustrated in Figure 4.2. $A_1 = A_0\alpha_D$ is the total number of feature data access to L2 due to their misses to L1; $A_2 = A_0\alpha_D\beta_D$ is the total number of feature data access to L3. $A_3 = A_0\alpha_D\beta_D\gamma_D$ is the total number of data access to memory.

Then the time cost divided by $A_0\delta_1$ is defined as

$$T = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D$$

where $T_D = 1+\alpha_D c_2+\alpha_D\beta_D c_3+\alpha_D\beta_D\gamma_D c_4$ and $T_S = 1+\alpha_S c_2+\alpha_S\beta_S c_3+\alpha_S\beta_S\gamma_S c_4$.

Since A_0 and δ_1 are constants, in the rest of the analysis, we focus on computing the above data access cost ratio T.

Notice that once data is brought from memory hierarchy, the arithmetic computing cost of all four methods is the same. Thus we just need to analyze and compare the data access cost ratio T for the four traversal methods. In practice, data access cost often weights more than arithmetic cost.

In this dissertation, we first present the analysis of cache performance for DSD and then list the result of DSDS, SDSD, and SDS as the case subdivision and cost derivation process are similar. Finally we describe an approximate scheme to select the best structure by taking advantages of the derived data access cost ratio for the four methods.

4.4 Cost Analysis and Comparison

4.4.1 Time Cost for DSD

4.4.1.1 Cases under consideration

```
1 for all vector blocks do
2     for i in all scorers do
3         for j in a vector block do
4             Update score for vector j with scorer i.
5         end
6     end
7 end
```

Algorithm 7: The program structure of DSD method.

Algorithm 7 lists the program control structure of DSD and a vector block contains d vectors. Once a scorer s_i is loaded to cache, it will be used by d vectors in the inner most loop. Then the next scorer s_{i+1} will go through the same d vectors. If we choose d properly such that d vectors fit in cache, we do not need to load them from memory for each scorer. Figure 4.3 illustrates how the cost of score computation could change when value d increases from 1 to n. The impact of d value on the cost is segmented with respect to the size of L1, L2, and L3. When

d is small, the d vectors can fit in L1 cache, and there is an advantage of reusing these d vectors within L1 cache. Thus d should be as large as possible. When d value becomes too big, the benefit of leveraging L1 cache decreases because d vectors may not fit in L1 any more and therefore the access cost can increase with larger d value. We can reason similarly when d vectors fit or do not fit in L2 and L3 caches.

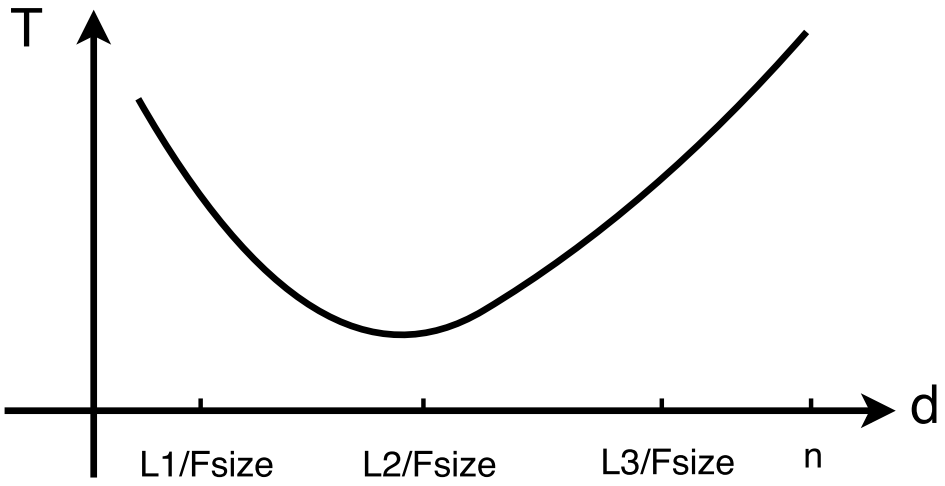


Figure 4.3: Performance under different values of d.

We will clarify the tradeoff of increasing d value when we derive a more concrete analysis. Let $Fsize$ be the average data size of each feature vector. Without introducing more symbols, we also let $L1, L2$ and $L3$ represent the size of L1 cache, L2 cache and L3 cache respectively in a formula expression. To assess the

impact of increasing d values, we divide the increasing range into four parts as illustrated in Figure 4.3.

- d vectors fits in L1 cache. Namely $d \leq \frac{L1}{Fsize}$
- d vectors do not fit in L1 cache, but fit in L2 cache. $\frac{L1}{Fsize} < d \leq \frac{L2}{Fsize}$
- d vectors do not fit in L2 cache, but fit in L3 cache. $\frac{L2}{Fsize} < d \leq \frac{L3}{Fsize}$
- d vectors exceed L3 cache and but fit in memory. $\frac{L3}{Fsize} < d \leq n$.

The cache access behavior of inner most loop in Algorithm 7 is affected by the average size of each scorer. For example, a larger scorer footprint leaves little space for L1 to host feature vectors. Figure 4.4 illustrates that we need to consider the following four scenarios and for each scenario, we need to further consider the four d range cases discussed above. Let $Ssize$ represent the average data size of each scorer and the four scenarios corresponding to the root branches in Figure 4.4 are defined as follows.

- **Scenario 1:** $Ssize \leq L1$. When a scorer can fit in L1 cache, there are four cases for the d vectors: the vector block fits in L1 cache, L2 cache, L3 cache or memory.
- **Scenario 2:** $L1 < Ssize \leq L2$. When a scorer size is between L1 and L2 cache sizes, the vector block can only fit in a higher level of cache (say L2

or L3 cache). Otherwise, old vectors will be kicked out from L1 cache by the scorer and the vector block could not stay in L1 cache. Thus there are only three cases for the d vectors as depicted in the second root branch of Figure 4.4: the vector block fits in L2 cache, L3 cache or memory.

- **Scenario 3:** $L2 < Ssize \leq L3$. When a scorer size is inbetween L2 and L3 cache size, there are two cases for the d vectors: the vector block fits in L3 cache or memory.

- **Scenario 4:** $Ssize > L3$. When a scorer cannot fit in L3 cache, the d feature vectors will not be able to fit in L3 cache also and they can only fit in memory.

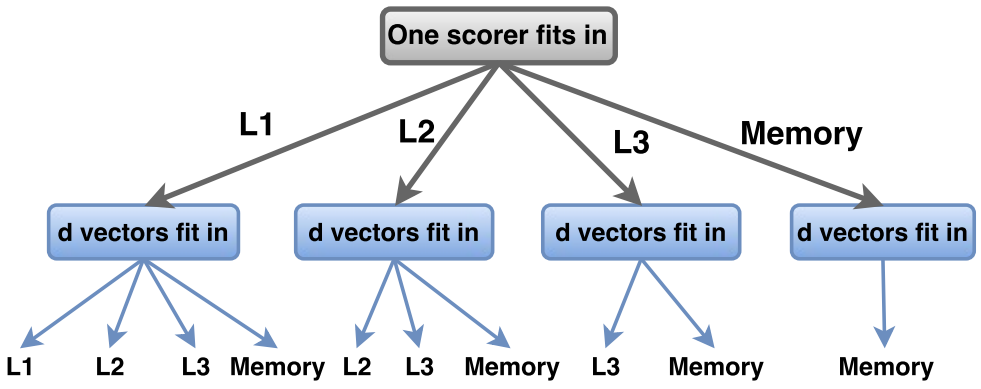


Figure 4.4: Range cases of d considered under different scenarios for DSD.

To simplify the analysis, we assume that m and n are sufficiently large so that m scorers do not fit in L3 cache, and also n feature vectors do not fit in L3 cache.

4.4.1.2 DSD under Scenario 1

Under Scenario 1, we first compute T_S as follows. Since each scorer is loaded once for the inner loop most and will re-used d times for computing the subscores for d vectors in the inner most loop. Then the L1 cache miss ratio $\alpha_S \approx 1/d$. If there is an L1 cache miss for a scorer, L2 cache miss and L3 cache miss can occur with a high chance because the unseen new scorer has not been used ever and thus it is fetched from memory. Thus $\beta_S \approx 1$ and $\gamma_S \approx 1$.

$$T_S \approx 1 + \frac{c_2}{d} + \frac{1}{d}(c_3 + c_4) \approx 1 + \frac{c_4}{d}$$

We shall estimate T_D under 4 different ranges of d values following Figure 4.3. We call these 4 range cases under DSD as DSD_i where $1 \leq i \leq 4$. The total cost ratio of accessing scorers for DSD is

$$T_{DSD_i} = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D \approx \eta + \frac{\eta c_4}{d} + T_D$$

where

$$T_D = 1 + c_2\alpha_i + \alpha_i\beta_i(c_3 + c_4\gamma_i) \tag{4.1}$$

and α_i, β_i and γ_i are cache miss rates for accessing feature vectors under range case i. Note that in differentiating these miss rate of different cases, we use script "i" instead of "D, i" in order to simplify the presentation. Table 4.1 summarizes the cost of DSD for Scenario 1 when each scorer fits in L1 cache on average.

Range Case DSD_1: d vectors fit in L1. Once a scorer is loaded to L1, the inner most loop load d feature vectors to L1 and these vectors stay and will be available in L1 when a new scorer is fetched to L1. Given m scorers, each feature vector in L1 is accessed m times and there is one 1 miss initially and the rest of $m-1$ accesses will hit L1. Thus the L1 cache miss ratio with respect to feature vectors is $\alpha_1 \approx 1/m$. If there is an L1 cache miss for a feature vector, there must be an L2 cache miss and L3 cache miss. Thus, $\beta_1 \approx 1$ and $\gamma_1 \approx 1$. Plugging into Equation 4.1, we get the total cost ratio:

$$T_D \approx 1 + \frac{c_2}{m} + \frac{1}{m} \cdot (c_3 + c_4).$$

Range case DSD_2: d vectors fits in L2. Once a scorer is loaded to L1, the inner most loop can load a feature vector and keep it at least at L2 when a new scorer is loaded. Given there are m scorers, $A_2/A_0 \approx 1/m$. Namely $\alpha_2\beta_2 = A_2/A_0 \approx 1/m$. Since L2 can hold d vectors needed for inner most loop, $A_2 \approx A_3$. Thus $\gamma_2 = A_3/A_2 \approx 1$.

$$T_D \approx 1 + \alpha_2 \cdot c_2 + \frac{1}{m} \cdot (c_3 + c_4).$$

Range case DSD_3: d vectors fit in L3. Once a scorer is loaded to L1, the inner most loop can load a feature vector and keep it at least at L3 when a new scorer is loaded. Given there are m scorers, $A_3/A_0 \approx 1/m$. Namely $\alpha_3\beta_3\gamma_3 = A_3/A_0 \approx 1/m$.

$$T_D \approx 1 + \alpha_3 \cdot c_2 + \alpha_3\beta_3 \cdot c_3 + \frac{1}{m} \cdot c_4.$$

Range case DSD_4: d vectors donot not fit in L3. In this case, $A_0 \approx A_1 \approx A_2 \approx A_3$. In this case, actually we put n documents in the inner loop. It's obvious to see that L1, L2 and L3's cache miss ratio are all 1 because comparing to the memory size, even L3 cache size is too small.

$\alpha_4 \approx 1$, $\beta_4 \approx 1$ and $\gamma_4 \approx 1$.

$$T_D \approx 1 + c_2 + c_3 + c_4.$$

Cases	d vectors fit in	$T_{DSD_i} = \eta T_s + T_D \approx$
DSD_1	L1	$\eta + \eta\frac{c_4}{d_1} + 1 + \frac{c_2+c_3+c_4}{m}$
DSD_2	L2	$\eta + \eta\frac{c_4}{d_2} + 1 + \alpha_2 c_2 + \frac{c_3+c_4}{m}$
DSD_3	L3	$\eta + \eta\frac{c_4}{d_3} + 1 + \alpha_3 c_2 + \alpha_3\beta_3 c_3 + \frac{c_4}{m}$
DSD_4	memory	$\eta + \eta\frac{c_4}{d_4} + 1 + c_2 + c_3 + c_4$

Table 4.1: Cost of DSD when 1 scorer fits in L1.

4.4.1.3 Other Scenarios of DSD

For Scenario 2 when a scorer fits in L2 on average, there are only 3 range cases: DSD_2, DSD_3, and DSD_4. L1 miss rate in T_D becomes 1 and T_S adds c_2 as

$T_S \approx 1 + c_2 + \frac{c_4}{d}$. For Scenario 3 where a scorer fits in L3, there are only 2 possible cases to consider: DSD_3 and DSD_4. L1 and L2 miss rates in T_D become 1 and T_S adds c_3 as $T_S \approx 1 + c_3 + \frac{c_4}{d}$. For Scenario 4 where a scorer fits memory only, there is one case to consider: DSD_4. Its T_D does not change while $T_S \approx 1 + c_4$.

4.4.2 Time Cost for SDS

4.4.2.1 SDS under Scenario 1

```
1 for all scorers blocks do
2     for i in a vector block do
3         for j in a scorer block do
4             Update score for vector i with scorer j.
5         end
6     end
7 end
```

Algorithm 8: The program structure of SDS method.

Under the condition that a feature vector can fit in L1, A scorer can fit in L1, L2, L3 or memory based on the scorer's size with regard to each memory hierachy's size. Different from 4.4.1.2, estimiated T_D in SDS is determined by the scorer block size s. For the inner most loop, a feature vector can be re-used s

times once it's loaded to compute its subscores. Then the L1 cache miss ratio $\alpha_D \approx 1/s$. If a feature vector has an L1 cache miss, L2 cache miss and L3 cache miss can occur with a high chance because an unseen feature vector has not been used ever and will be fetched from memory. Therefore, $\beta_D \approx 1$ and $\gamma_D \approx 1$.

$$T_D \approx 1 + \frac{1}{s}(c_2 + c_3 + c_4) \approx 1 + \frac{c_4}{s}$$

We shall estimate T_S under 4 different ranges of s values following the similar logic with Figure 4.3. We call these 4 range cases under SDS as SDS_i where $1 \leq i \leq 4$. The total cost ratio of accessing scorers for SDS is

$$T_{SDS_i} = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D \approx \eta T_S + (1 + \frac{c_4}{s})$$

where

$$T_S = 1 + c_2\alpha_i + \alpha_i\beta_i(c_3 + \gamma_i c_4) \tag{4.2}$$

and α_i, β_i and γ_i are cache miss rates for accessing scorers under range case i. Note that in differentiating these miss rate of different cases, we use script "i" instead of "S, i" in order to simplify the presentation. Table 4.2 summarizes the cost of SDS under Scenario1 when each feature vector fits in L1 cache on average.

Range Case SDS_1: s scorers fit in L1. For each feature vector, the inner most loop loads s scorers to L1 and these scorers will stay and be available in L1 when a new feacture vector is fetched to L1. Given n feature vectors, each scorer

in L1 is accessed n times and there is only 1 L1 cache miss initially and the rest of $n-1$ accesses will hit L1. Thus the L1 cache miss ratio of scorers is $\alpha_1 \approx 1/n$. If there is an L1 cache miss for a feature vector, there must be an L2 cache miss and L3 cache miss. Thus, $\beta_1 \approx 1$ and $\gamma_1 \approx 1$. In addition, we can ignore c_2 and c_3 since $c_2 << c_3 << c_4$. Plugging into Equation 4.2, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{n} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{n} \cdot c_4.$$

Range case SDS_2: s scorers fit in L2. For each feature vector, the inner most loop loads s scorers to L2 and these scorers will stay and be always available in L2 when a new feacture vector is fetched to L1. α_1 can also be quantified since s scorers fit in L2 and if one scorer is loaded from L2 to L1, all nodes in the tree could be visited based on tree traversal path. Therefore, after all s scorers being visited once, the second visit will start and the first scorer visited before must be swapped out of L1 since the capacity of L1 is less than that of L2. Thus, we can conclude that $\alpha_1 = 1$. In addition, we know that given n feature vectors, $A_2/A_0 \approx 1/n$. Namely $\alpha_2\beta_2 = A_2/A_0 \approx 1/n$. Since L2 can only hold s scorers for inner most loop, every time accessing L3 must trigger one access to memory. Therefore, $A_2 \approx A_3$ and $\gamma_2 = A_3/A_2 \approx 1$.

$$T_S \approx 1 + c_2 + \frac{1}{n} \cdot (c_3 + c_4) \approx 1 + c_2.$$

Range case SDS_3: s scorers fit in L3. For each feature vector, the inner most loop loads s scorers to L3 and these scorers will stay and be always available in L3 when a new feacture vector is fetched to L1. $\alpha_3 \approx \alpha_3\beta_3 \approx 1$. However, given there are n feature vectors, $A_3/A_0 \approx 1/n$. Namely $\alpha_3\beta_3\gamma_3 = A_3/A_0 \approx 1/n$.

$$T_S \approx 1 + c_2 + c_3 + \frac{1}{n} \cdot c_4 \approx 1 + c_3.$$

Range case SDS_4: s scorers can not fit in L3. In this case, $A_0 \approx A_1 \approx A_2 \approx A_3$. In this case, actually we put m scorers in the inner loop. It's obvious to see that L1, L2 and L3's cache miss ratio are always 1 because comparing to the memory size, even L3 cache size is too small.

$\alpha_4 \approx 1$, $\beta_4 \approx 1$ and $\gamma_4 \approx 1$.

$$T_S \approx 1 + c_2 + c_3 + c_4 \approx 1 + c_4.$$

4.4.2.2 Other Scenarios of SDS

Other scenarios include 3 situations: 1) one feature vector fits in L2; 2) one feature vector fits in L3; 3) one feature vector fits in memory. For the scenario that one feature vector fits in L2, we only need to consider SDS_2, SDS_3 and SDS_4. Formula T_S is the same as table 4.2 while T_D adds c_2 and becomes: $T_D = 1 + c_2 + \frac{c_4}{s}$. For the scenario when a feature vector fits in L3, only 2 cases need to be considered: SDS_3 and SDS_4. Formula T_S still stays the same while

Cases	s scorers fit in	$T_{SDS_i} = T_D + \eta T_s \approx$
SDS_1	L1	$1 + \frac{c_4}{s_1} + \eta + \eta\frac{c_4}{n}$
SDS_2	L2	$1 + \frac{c_4}{s_2} + \eta + \eta c_2$
SDS_3	L3	$1 + \frac{c_4}{s_3} + \eta + \eta c_3$
SDS_4	memory	$1 + \frac{c_4}{s_4} + \eta + \eta c_4$

Table 4.2: Cost of SDS when 1 feature vector can fit in L1.

T_D adds c_3 ($c_3 >> c_2$ such that c_2 is dropped) and becomes $T_D = 1 + c_3 + \frac{c_4}{s}$. For the last scenario, we only consider SDS_4 in which T_S is the same while T_D changes to $T_D = 1 + c_4$.

4.4.3 Time Cost for DSDS

4.4.3.1 DSDS under Scenario 1

A feature vector can fit in L1 and a scorer vector can fit in L1, L2, L3 or memory based on the scorer's size with regard to each memory hierachy's size. Different from 4.4.1.2 and 4.4.2, DSDS need to consider the size of d and s togother with regard to different memory level's size. Therefore, instead of 4 range cases in DSD and SDS, there are 10 cases to be analyzed when estimating T_S and T_D. We call these 10 range cases under DSDS as DSD_iS_j where $1 \leq j \leq i \leq 4$.

```
1 for all vector blocks do
2     for all scorers blocks do
3         for i in a vector block do
4             for j in a scorer block do
5                 Update score for vector i with scorer j.
6             end
7         end
8     end
9 end
```

Algorithm 9: The program structure of DSDS method.

The total cost ratio of accessing scorers for DSDS is

$$T_{DSD_iS_j} = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D \approx \eta + \eta \frac{c_4}{d} + T_D$$

where

$$T_D = 1 + c_2\alpha_i + \alpha_i\beta_i(c_3 + c_4\gamma_i) \tag{4.3}$$

$$T_S = 1 + c_2\alpha_j + \alpha_j\beta_j(c_3 + c_4\gamma_j) \tag{4.4}$$

α_i, β_i and γ_i are cache miss rates for accessing feature vectors; α_j, β_j and γ_j are cache miss rates for accessing scorers within the range $1 \leq j \leq i \leq 4$. Note that in differentiating these miss rate of different cases, we use script "i" instead of "D, i" and "j" instead of "S, j" in order to simplify the presentation. Table 4.3

summarizes the cost of DSDS under Scenario1 when each feature vector fits in L1 cache on average.

For range cases of DSD_iS_1 (s scorers always fit in L1), we first compute T_S as follows. Since each scorer block is loaded once for the inner loop most and will be re-used d times for computing the subscores for d vectors in a feature vector block. Then the L1 cache miss ratio $\alpha_S \approx 1/d$. If there is an L1 cache miss for a scorer, L2 cache miss and L3 cache miss can occur with high possibility because the unseen new scorer has not been used ever and it will be fetched from memory. Thus $\beta_S \approx 1$ and $\gamma_S \approx 1$.

$$T_S \approx 1 + \frac{1}{d}(c_2 + c_3 + c_4) \approx 1 + \frac{c_4}{d}$$

Range Case DSD_1S_1: s scorers fit in L1 and each feature vector block fits in L1 as well. For each feature vector in one feature block, the inner most loop loads s scorers to L1. The second inner loop is responsible to control the load of the next scorer block in the fourth inner loop, but the third inner loop still uses the same feature block. Therefore, each feacture vector will be accessed for m times, and only the first access will be missed in L1. So $\alpha_1 \approx 1/m$. If there is an L1 cache miss for a feature vector, there is very high possibility with an L2 cache miss and L3 cache miss. Thus, $\beta_1 \approx 1$ and $\gamma_1 \approx 1$. In addition, we can ignore c_2 and c_3 since $c_2 << c_3 << c_4$. Plugging into Equation 4.3, we get the total cost

ratio:

$$T_D \approx 1 + \frac{1}{m} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{m} \cdot c_4.$$

Range case DSD_2S_1: d feature vectors fit in L2 and s scorers fit in L1. Once we load one scorer block (s scorers) into L1, each feature vector loaded by the third inner loop will be accessed s times, and only the first time will be missed in L1. Thus, we can estimate $\alpha_1 \approx 1/s$. Besides, since one feature block (d feature vectors) fit in L2, each feature vector in L2 will be accessed m times in total with only the first time being L2 miss. Therefore, L2 cache miss ratio for each feature vector, that is $\alpha_1\beta_1$, is estimated as $1/m$ and $\gamma_1 = A_3/A_2 \approx 1$.Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot c_2 + \frac{1}{m} \cdot (c_3 + c_4).$$

Furthermore, since $s << m$, $\frac{1}{s} \cdot c_2$ will be much greater than $\frac{1}{m} \cdot (c_3 + c_4)$, we can make the formula above more concise and easy to analysis:

$$T_D \approx 1 + \frac{1}{s} \cdot c_2.$$

Range case DSD_3S_1: d feature vectors fit in L3 and s scorers fit in L1. Once we load one scorer block (s scorers) into L1, each feature vector loaded by the third inner loop will be accessed s times, and only the first time will be missed in L1 and L2 under the condition that d feature vectors can only fit into L3. Thus, we

can estimate $\alpha_1 \approx 1/s$ and $\alpha_1\beta_1 \approx 1/s$. Besides, since one feature block (d feature vectors) fit in L3, each feature vector in L3 will be accessed m times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each feature vector, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/m$. In addition, $\frac{1}{s}\cdot(c_2+c_3) >> \frac{1}{m}\cdot c_4$ and $c_2 << c_3$, we can get the total cost ratio as follows with regard to Equation 4.3:

$$T_D \approx 1 + \frac{1}{s}\cdot(c_2+c_3) + \frac{1}{m}\cdot c_4 \approx 1 + \frac{1}{s}\cdot c_3.$$

Range case DSD_4S_1: d feature vectors cannot fit in L3 and s scorers still fit in L1. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/s$.

Referring to Equation 4.3, we get the total cost ratio with the condition that $c_2 << c_3 << c_4$:

$$T_D \approx 1 + \frac{1}{s}\cdot(c_2+c_3+c_4) \approx 1 + \frac{1}{s}\cdot c_4.$$

For range cases of DSD_iS_2 (s scorers always fit in L2 and i in range [2, 4]), we first compute T_S as follows. The fourth loop will traverse all the s scorers for one feature vector and for the next feature vector, it will re-visit the same s scorers in order. Since these s scorers can only fit in L2, it means that the second time visiting the same scorer, one L1 miss will happen. Therefore, we can estimate $\alpha_S \approx 1$. Since one scorer block always fit into L2, for d iterations in the third loop, the fourth loop only need to load one scorer block once and for all the

following d-1 iterations, accesses of this scorer block will hit on L2. Therefore, we can conclude that $\alpha_S\beta_S \approx \frac{1}{d}$ and $\gamma_S \approx 1$.

$$T_S \approx 1 + c_2 + \frac{1}{d}(c_3 + c_4) \approx 1 + c_2 + \frac{1}{d} \cdot c_4$$

Range Case DSD_2S_2: d feature vectors fit in L2 and s scorers fit in L2 as well.

Once we load one scorer block (s scorers) into L2, each feature vector loaded by the third inner loop will be accessed s times, and only the first time will be missed in L1. Thus, we can estimate $\alpha_1 \approx 1/s$. Besides, since one feature block (d feature vectors) fit in L2, each feature vector in L2 will be accessed m times in total with only the first time being L2 miss. Therefore, L2 cache miss ratio for each feature vector, that is $\alpha_1\beta_1$, is estimated as $1/m$ and $\gamma_1 = A_3/A_2 \approx 1$. Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot c_2 + \frac{1}{m} \cdot (c_3 + c_4) \approx 1 + \frac{1}{c} \cdot c_2.$$

Range Case DSD_3S_2: d feature vectors fit in L3 and s scorers fit in L2. Once we load one scorer block (s scorers) into L2, each feature vector loaded by the third inner loop will be accessed s times, and only the first time will be missed in L1 and L2 under the condition that d feature vectors can only fit into L3. Thus, we can estimate $\alpha_1 \approx \alpha_1\beta_1 \approx 1/s$. Besides, since one feature block (d feature

vectors) fit in L3, each feature vector in L3 will be accessed m times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each feature vector, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/m$. Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot (c_2 + c_3) + \frac{1}{m} \cdot c_4 \approx 1 + \frac{1}{s} \cdot c_3.$$

Range Case DSD_4S_2: d vectors fit in memory and s scorers fit in L2. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/s$.

Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{s} \cdot c_4.$$

For range cases of DSD_iS_3 (s scorers always fit in L3 and i in range [3, 4]), we first compute T_S as follows. The fourth loop will traverse all the s scorers for one feature vector and for the next feature vector, it will re-visit the same s scorers in order. Since these s scorers can only fit in L3, it means that the second time visiting the same scorer, one L1 miss and one L2 miss will happen. Therefore, we can estimate $\alpha_S \approx \alpha_S\beta_S \approx 1$. Since one scorer block always fit into L3, for d iterations in the third loop, the fourth loop only need to load one scorer block once and for all the following d-1 iterations, accesses of this scorer block will hit on L3. Therefore, we can conclude that $\alpha_S\beta_S\gamma_S \approx \frac{1}{d}$.

$$T_S \approx 1 + c_2 + c_3 + \frac{1}{d} \cdot c_4 \approx 1 + c_3 + \frac{1}{d} \cdot c_4$$

Range Case DSD_3S_3: d feature vectors fit in L3 and s scorers fit in L3 as well. Once we load one scorer block (s scorers) into L3, each feature vector loaded by the third inner loop will be accessed s times, and only the first time will be missed in L1 and L2 under the condition that d feature vectors can only fit into L3. Thus, we can estimate $\alpha_1 \approx \alpha_1\beta_1 \approx 1/s$. Besides, since one feature block (d feature vectors) fit in L3, each feature vector in L3 will be accessed m times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each feature vector, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/m$. Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot (c_2 + c_3) + \frac{1}{m} \cdot c_4 \approx 1 + \frac{1}{s} \cdot c_3.$$

Range Case DSD_4S_3: d feature vectors only fit in memory and s scorers fit in L3. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/s$.

Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{s} \cdot c_4$$

Range Case DSD_4S_4: d feature vectors and s scorers cannot fit in L3. That is, all feature vectors and scorers stay in memory. First we estimate T_S. Since m

scorers fit in memory only, every time loading one scorer block will encounter L1, L2 and L3 cache miss. therefore, $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1$ (here α_1 denotes the L1 cache miss ratio for scorers, so as to others). We have T_S as:

$$T_S \approx 1 + (c_2 + c_3 + c_4) \approx 1 + c_4$$

Although every time accessing one feature block will have L1, L2 and L3 misses, each block will be reused for s times before being swapped out of the caches. Therefore, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/s$ (here α_1 denotes the L1 cache miss ratio for feature vectors, so as to others). Referring to Equation 4.3, we get the total cost ratio:

$$T_D \approx 1 + \frac{1}{s} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{s} \cdot c_4$$

4.4.3.2 Other Scenarios of DSDS

There are 3 other scenarios for DSDS with regard to one feature vector's size: 1) one feature vector fits in L2; 2) one feature vector fits in L3; 3) one feature vector fits in memory. For the scenario that one vector fits in L2, we only need to consider DSD_2S_2, DSD_3S_2, DSD_4S_2, DSD_3S_3, DSD_4S_3 and DSD_4S_4. Formula T_S is the same as 4.3 while T_D adds c_2 to its formula in the table since it cannot fit to L1 and every access has L1 cache miss. For the scenario when a feature vector fits in L3, only 3 cases need to be considered: DSD_3S_3, DSD_4S_3, DSD_4S_4. Formula

Cases	d vectors fit in	s scorers fit in	$T_{DSD_iS_j} = T_D + \eta T_S \approx$
DSD_1S_1	L1	L1	$1 + \frac{c_4}{m} + \eta + \eta\frac{c_4}{d_1}$
DSD_2S_1	L2	L1	$1 + \frac{c_2}{s_1} + \eta + \eta\frac{c_4}{d_2}$
DSD_3S_1	L3	L1	$1 + \frac{c_3}{s_1} + \eta + \eta\frac{c_4}{d_3}$
DSD_4S_1	memory	L1	$1 + \frac{c_4}{s_1} + \eta + \eta\frac{c_4}{d_4}$
DSD_2S_2	L2	L2	$1 + \frac{c_2}{s_2} + \eta + \eta c_2 + \eta\frac{c_4}{d_2}$
DSD_3S_2	L3	L2	$1 + \frac{c_3}{s_2} + \eta + \eta c_2 + \eta\frac{c_4}{d_3}$
DSD_4S_2	memory	L2	$1 + \frac{c_4}{s_2} + \eta + \eta c_2 + \eta\frac{c_4}{d_4}$
DSD_3S_3	L3	L3	$1 + \frac{c_3}{s_3} + \eta + \eta c_3 + \eta\frac{c_4}{d_3}$
DSD_4S_3	memory	L3	$1 + \frac{c_4}{s_3} + \eta + \eta c_3 + \eta\frac{c_4}{d_4}$
DSD_4S_4	memory	memory	$1 + \frac{c_4}{s_4} + \eta + \eta c_4$

Table 4.3: Cost of DSDS when 1 feature vector fits in L1.

T_S still stays the same while T_D adds c_3 ($c_3 >> c_2$ such that c_2 is dropped) to its part in table 4.3. For the last scenario, we only consider DSD_4S_4 in which T_S is the same while T_D changes to $T_D = 1 + c_4$.

4.4.4 Time Cost for SDSD

4.4.4.1 SDSD under Scenario 1

```
1 for all scorers blocks do
2     for all vectors blocks do
3         for i in a scorer block do
4             for j in a vector block do
5                 Update score for vector j with scorer i.
6             end
7         end
8     end
9 end
```

Algorithm 10: The program structure of SDSD method.

For this scenario, a scorer fits in L1. A feature vector can fit in L1, L2, L3 or memory based on the feature vector's size with regard to each memory hierachy's size.Similar to 4.4.3, SDSD needs to consider the size of s and d togother with

regard to different memory level's size. Therefore, there are also 10 cases to be analyzed when estimating T_S and T_D. We call these 10 range cases under SDSD as SDS_iD_j where $1 \leq j \leq i \leq 4$. Table 4.4 summarizes the cost of SDSD under Scenario1 when each scorer fits in L1 cache on average.

The total cost ratio of accessing scorers for SDSD is

$$T_{SDS_iD_j} = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D \approx \eta T_S + 1 + \frac{c_4}{s}$$

where

$$T_S = 1 + c_2\alpha_i + \alpha_i\beta_i(c_3 + c_4\gamma_i) \tag{4.5}$$

$$T_D = 1 + c_2\alpha_j + \alpha_j\beta_j(c_3 + c_4\gamma_j) \tag{4.6}$$

α_i, β_i and γ_i are cache miss rates for accessing scorers and α_j, β_j and γ_j are cache miss rates for accessing feature vectors within the range $1 \leq j \leq i \leq 4$. Note that in differentiating these miss ratio of different cases, we use symbol "i" instead of "S, i" and "j" instead of "D, j" in order to simplify the presentation.

For range cases of SDS_iD_1 (d feature vectors always fit in L1), we first compute T_D as follows. Since each feature vector block is loaded once for the inner most loop and will be re-used s times for computing the subscores for d vectors in a feature vector block. Then the L1 cache miss ratio $\alpha_D \approx 1/s$. If there is an L1 cache miss for a feature vector, L2 cache miss and L3 cache miss can occur with

a high chance because the unseen new feature vector has not been used ever and thus it is fetched from memory. Thus $\beta_D \approx 1$ and $\gamma_D \approx 1$.

$$T_D \approx 1 + \frac{1}{s}(c_2 + c_3 + c_4) \approx 1 + \frac{c_4}{s}$$

We will calculate T_S in the following 4 range cases.

Range Case SDS_1D_1: d feature vectors fit in L1 and each score block fits in L1 as well. For each scorer in one scorer block, the inner most loop loads d feature vectors to L1. The second inner loop is responsible to control the load of the next feature vector block in the fourth inner loop, but the third inner loop still uses the same scorer block. Therefore, each score will be accessed for n times, and only the first access will be missed in L1. So $\alpha_1 \approx 1/n$. If there is an L1 cache miss for a feature vector, there is very high possibility with an L2 cache miss and L3 cache miss. Thus, $\beta_1 \approx 1$ and $\gamma_1 \approx 1$. Plugging into Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{n} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{n} \cdot c_4.$$

Range case SDS_2D_1: s scorers fit in L2 and d feature vectors fit in L1. Once we load one feature vector block (d feature vectors) into L1, each scorer loaded by the third inner loop will be accessed d times, and only the first time will be missed in L1. Thus, we can estimate $\alpha_1 \approx 1/d$. Besides, since one scorer block (s scorers) fit in L2, each scorer in L2 will be accessed n times in total with only the

first time being L2 miss. Therefore, L2 cache miss ratio for each scorer, that is $\alpha_1\beta_1$, is estimated as $1/n$ and $\gamma_1 = A_3/A_2 \approx 1$.Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot c_2 + \frac{1}{n} \cdot (c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_2.$$

Range case SDS_3D_1: s scorers fit in L3 and d feature vectors fit in L1. Once we load one feature vector block (d feature vectors) into L1, each scorer loaded by the third inner loop will be accessed d times, and only the first time will be missed in L1 and L2 under the condition that s scorers can only fit into L3. Thus, we can estimate $\alpha_1 \approx 1/d$ and $\alpha_1\beta_1 \approx 1/d$. Besides, since one scorer block (s scorers) fit in L3, each scorer in L3 will be accessed n times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each feature vector, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/n$. Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3) + \frac{1}{n} \cdot c_4 \approx 1 + \frac{1}{d} \cdot c_3.$$

Range case SDS_4D_1: s scorers cannot fit in L3 and d feature vectors still fit in L1. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/d$.

Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_4.$$

For range cases of SDS_iD_2 (d feature vectors always fit in L2 and i in range [2, 4]), we first compute T_D as following. The fourth loop will traverse all the d feature vectors for one scorer and for the next scorer, it will re-visit the same d feature vectors in order. Although these d feature vectors only fit in L2, there is still possibility that when the scorer vector is small and only a small portion of one feature vector is visited, one L1 miss may not happen for next visit. Therefore, we can't estimate the value of α_D. However, since one feature vector block always fit into L2, for s iterations in the third loop, the fourth loop only need to load one feature vector block once and for all the following s-1 iterations, accesses of this feature vector block will hit in L2. Therefore, we can conclude that $\alpha_D\beta_D \approx \frac{1}{s}$ and $\gamma_D \approx 1$.

$$T_D \approx 1 + \alpha_D c_2 + \frac{1}{s} \cdot (c_3 + c_4) \approx 1 + \alpha_D c_2 + \frac{1}{s} \cdot c_4.$$

Then we will calculate T_S in the following 3 range cases.

Range Case SDS_2D_2: s scorers fit in L2 and d feature vectors fit in L2 as well.

Once we load one feature vector block (d feature vectors) into L2, each scorer loaded by the third inner loop will be accessed d times, and only the first time

will be missed in L1. Thus, we can estimate $\alpha_1 \approx 1/d$. Besides, since one scorer block (s scorers) fit in L2, each scorer in L2 will be accessed n times in total with only the first time being L2 miss. Therefore, L2 cache miss ratio for each scorer, that is $\alpha_1\beta_1$, is estimated as $1/n$ and $\gamma_1 = A_3/A_2 \approx 1$. Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot c_2 + \frac{1}{n} \cdot (c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_2.$$

Range Case SDS_3D_2: s vectors fit in L3 and d feature vectors fit in L2. Once we load one feature block (d feature vectors) into L2, each scorer loaded by the third inner loop will be accessed d times, and only the first time will be missed in L1 and L2 under the condition that s scorers can only fit into L3. Thus, we can estimate $\alpha_1 \approx \alpha_1\beta_1 \approx 1/d$. Besides, since one scorer block (s scorers) fit in L3, each scorer in L3 will be accessed n times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each feature vector, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/n$. Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3) + \frac{1}{n} \cdot c_4 \approx 1 + \frac{1}{d} \cdot c_3.$$

Range Case SDS_4D_2: s scorers fit in memory and d feature vectors still fit in L2. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/d$.

Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_4.$$

For range cases of SDS_iD_3 (d feature vectors always fit in L3 and i in range [3, 4]), we first compute T_D as follows. The fourth loop will traverse all the d feature vectors for one scorer and for the next scorer, it will re-visit the same d feature vectors in order. Although these d feature vectors only fit in L3 and it seems visiting a feature vector will always have L1 and L2 miss, there is still possibility that when the scorer vector is small and only a small portion of one feature vector is visited, one L1 and L2 miss may not happen for next visit. Therefore, we can't estimate the value of α_D and $\alpha_D\beta_D$. However, since one feature block always fit into L3, for s iterations in the third loop, the fourth loop only need to load one feature block once and for all the following s-1 iterations, accesses of this scorer block will hit on L3. Therefore, we can conclude that $\alpha_D\beta_D\gamma_D \approx \frac{1}{s}$.

$$T_D \approx 1 + \alpha_D c_2 + \alpha_D\beta_D c_3 + \frac{1}{s} \cdot c_4.$$

Range Case SDS_3D_3: s scores fit in L3 and d vectors fit in L3 as well. Once we load one feature vector block (d vectors) into L3, each scorer loaded by the third inner loop will be accessed d times, and only the first time will be missed in L1 and L2 under the condition that s scorers can only fit into L3. Thus, we

can estimate $\alpha_1 \approx \alpha_1\beta_1 \approx 1/d$. Besides, since one scorer block (s scorers) fit in L3, each scorer in L3 will be accessed n times in total with only the first time being L3 miss. Therefore, L3 cache miss ratio for each scorer, that is $\alpha_1\beta_1\gamma_1$, is estimated as $1/n$. Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3) + \frac{1}{n} \cdot c_4 \approx 1 + \frac{1}{d} \cdot c_3.$$

Range Case SDS_4D_3: s scorers fit in memory and d feature vectors fit in L3. In this case, $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/s$.

Referring to Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_4$$

Range Case SDS_4D_4: s scorers fit in memory and d feature vectors fit in memory as well. First we estimate T_D. Since d feature vectors fit in memory only, every time loading one feature vectors block will enconter L1, L2 and L3 cache miss. therefore, $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1$ (here α_1 denotes the L1 cache miss ratio for scorers, so as to others). We have T_D as:

$$T_D \approx 1 + (c_2 + c_3 + c_4) \approx 1 + c_4$$

Then let's check T_S. $A_0 \approx A_1 \approx A_2$ and $\alpha_1 \approx \alpha_1\beta_1 \approx \alpha_1\beta_1\gamma_1 \approx 1/d$ (here α_1 denotes the L1 cache miss ratio for scorers, so as to others). Referring to

Equation 4.5, we get the total cost ratio:

$$T_S \approx 1 + \frac{1}{d} \cdot (c_2 + c_3 + c_4) \approx 1 + \frac{1}{d} \cdot c_4$$

Cases	s scorers fit in	d vectors fit in	$T_{SDS_iD_j} = T_D + \eta T_S \approx$
SDS_1D_1	L1	L1	$1 + \frac{c_4}{s_1} + \eta + \eta\frac{c_4}{n}$
SDS_2D_1	L2	L1	$1 + \frac{c_4}{s_2} + \eta + \eta\frac{c_2}{d_1}$
SDS_3D_1	L3	L1	$1 + \frac{c_4}{s_3} + \eta + \eta\frac{c_3}{d_1}$
SDS_4D_1	memory	L1	$1 + \frac{c_4}{s_4} + \eta + \eta\frac{c_4}{d_1}$
SDS_2D_2	L2	L2	$1 + \alpha_{2,2}c_2 + \frac{c_4}{s_2} + \eta + \eta\frac{c_2}{d_2}$
SDS_3D_2	L3	L2	$1 + \alpha_{3,2}c_2 + \frac{c_4}{s_3} + \eta + \eta\frac{c_3}{d_2}$
SDS_4D_2	memory	L2	$1 + \alpha_{4,2}c_2 + \frac{c_4}{s_4} + \eta + \eta\frac{c_4}{d_2}$
SDS_3D_3	L3	L3	$1 + \alpha_{3,3}c_2 + \alpha_{3,3}\beta_{3,3}c_3 + \frac{c_4}{s_3} + \eta + \eta\frac{c_3}{d_3}$
SDS_4D_3	memory	L3	$1 + \alpha_{4,3}c_2 + \alpha_{4,3}\beta_{4,3}c_3 + \frac{c_4}{s_4} + \eta + \eta\frac{c_4}{d_3}$
SDS_4D_4	memory	memory	$1 + c_4 + \eta + \eta\frac{c_4}{s_4}$

Table 4.4: Cost of SDSD when 1 score fits in L1.

4.4.4.2 Other Scenarios of SDSD

For the scenario that one scorer fits in L2, we only need to consider SDS_2D_2, SDS_3D_2, SDS_4D_2, SDS_3D_3, SDS_4D_3 and SDS_4D_4. Since L1 cache's capacity is not enough for one scorer, accessing a feature vector for every scorer will always need to fetch data from L2. Namely, accessing the feature vector always has L1 cache miss. Therefore, α in T_D is 1. Besides, T_S adds c_2 to its formula in the table since it cannot fit to L1 and every access has L1 cache miss. For the scenario when a scorer fits in L3, only 3 cases need to be considered: SDS_3D_3, SDS_4D_3, SDS_4D_4. α and β both become 1 in Formula T_D while T_S adds c_3 ($c_3 >> c_2$ such that c_2 is dropped) to its part in table 4.4. For the last scenario, DSD_4S_4 is considered in which T_D is the same while T_S changes to $T_S = 1 + c_4$.

4.4.5 Cost Comparison of the Four Methods

There is a total of 28 range cases considered in these four methods: DSD_i, SDS_i, DSD_iS_j, and SDS_iD_j where $1 \leq i, j \leq 4$. The cost results of these 28 cases can be used in the following two aspects.

- Identify the approximated optimum selection from 28 cases instead of exhaustive search of all combinations. The selection algorithm goes through 28 cases and runs the average time performance sampling through a benchmark of q queries.

From Tables 4.1, Tables 4.2, Tables 4.3, and Tables 4.4, increasing d or s under its range limit decreases the time cost. Thus d and s should be chosen to be as large as possible. On the other hand, scorers and vectors share each level of cache and we set a constraint that vectors and scorers accessed in the inner most loop of DSD and SDS or in the two inner most loops of DSDS and SDSD fit in the corresponding cache. For example, as a midpoint approximation, we let d vectors occupy upto half of each cache level and s scorers occupy upto half of each cache level. Namely, for $1 \leq i \leq 3$, d and s are chosen for each range case as: $d_i = \frac{0.5Li}{Fsize}$, $s_i = \frac{0.5Li}{Ssize}$. As all n vectors and m scorers fit in memory, $d_4 = n$ and $s_4 = m$.

- Narrow the search scope when characteristics of a dataset or targeted machine architecture is given because many of 28 cases may be eliminated. That facilitates the reduction of search space and allows more sampling points at each range case selected as long as time complexity permits. For example, the above midpoint sampling for each range case after elimination can be expanded as follows. Since each scorer may only access a fraction of a feature vector and a fraction of the scorer data structure for computation, we choose sampling points as $d_i = \frac{0.5Li}{\mu Fsize}$, $s_i = \frac{0.5Li}{\mu Ssize}$. Coefficient μ represents the average data usage of a vector or a scorer during computation and for example, we sample more points with μ as 1, 0.75, 0.5, and 0.25.

To illustrate the second point above, we show that the following proposition is true and can narrow the search scope from 28 to 4 range cases.

Proposition 4.4.1. *When each feature vector fits in L1 and each score fits in L1 on average, and* $\frac{\eta c_4}{d_2} \ll 1$, $\frac{c_4}{s_2} \ll 1$, *the candidates with the lowest access cost are among range cases* DSD_2, DSD_2S_1, SDS_2D_2, *and* SDS_2D_1.

A proof is listed in 4.4.7. Yahoo!, MS, and MQ datasets discussed in Section 4.5 fall into the condition of this proposition when each regression tree used is not too big (e.g. containing upto 50 leaves). The range of d and s values for these datasets is listed in Table 4.6 and Table 4.5 of Section 4.5. When a regression tree contains 150 leaves, ratio c_4/s_2 is getting close to 1, cases SDS_3D_1 and SDS_3D_2 can be competitive as a best candidate. Thus with such a condition, we can search for 6 cases instead of 28 cases.

In summary, a guided sampling scheme conducts the following steps. 1) Identify data and architecture parameters. When possible, apply Proposition 1 or its variation to eliminate some of 28 range cases from the cost analysis of DSD, SDS, DSDS, and SDSD. 2) For each of selected range cases, choose blocking factor d_i and s_i under a constraint that vectors and scorers accessed in the inner most loop of DSD and SDS or in the two inner most loops of DSDS and SDSD fit in the corresponding level of cache. One approach is to choose $d_i = \frac{0.5Li}{\mu Fsize}$ $s_i = \frac{0.5Li}{\mu Ssize}$ with a number of sampled μ values. 3) Run and collect the average query response

time with m scorers and n vectors from each sampled case. Select the case and parameter setting with the lowest response time. The total complexity of this scheme with q test queries is $O(m * n * q)$.

4.4.6 Discussions

Integration with the QuickScorer method. When the ensemble computation uses the original computing algorithm for gradient boosted regression trees (e.g. [48, 24]), the main data structure of each scorer is a tree. To use the BWQS algorithm [76], we treat each scorer as the application of QS on a block of trees. The following parameters are involved: the size of a scorer changes when different partitioning is adopted while the number of inner-loop scorers (s) and inner-loop vectors (d) can vary too for different blocking methods. Thus we add a partitioning search loop on the top of the aforementioned comparison and sampling scheme to select the best partitioning.

Batched query processing. When the ensemble score computation is used for query processing where n is small, d value of the inner most loop limited by n can be insufficient to explore the cache locality and the effectiveness of blocking degrades. When batch processing is allowed, we can boost the cache utilization by processing feature vectors from multiple queries in fast cache, which essentially raises n values. One application of such batched processing is to conduct an offline

experiment to assess the ranking performance of an algorithm in answering a large number of queries and there is no need to output ranking results immediately.

For an online ranking application, the ranking results need to be produced promptly. While reaching a high throughput, batching a large number of queries can increase the average waiting time of batched queries and affect the response time. With this constraint in mind, we set a limit on the largest waiting time allowed in choosing a batch size for a higher throughout with a modest increase of response time.

4.4.7 Proof for Proposition 1

For each of DSD, SDS, DSDS, and SDSD methods, we eliminate its range cases that do not qualify for the best candidate as follows.

- For DSD cases listed in Table 4.1, case DSD_4 is excluded from the best case list as term c_4 in DSD_4 cost dominates the weight. Thus $T_{DSD_4} > T_{DSD_3}$, $T_{DSD_4} > T_{DSD_2}$, and $T_{DSD_4} > T_{DSD_1}$. We also drop DSD_1 because $T_{DSD_1} > T_{SDS_2D_1}$.

  Now we compare DSD_2 and DSD_3. Since $\frac{\eta c_4}{d_2} \ll 1$, $\frac{\eta c_4}{d_3} \ll 1$, and these two terms can be dropped approximately from the cost expressions. Note that $\alpha_3\beta_3 = \frac{1}{m\gamma_3} \geq \frac{1}{m}$, and c_3 is much larger than c_2. Also $\alpha_2 < \alpha_3$ since the inner most loop of DSD_2 accesses less vectors. This makes $T_{DSD_3} > T_{DSD_2}$.

- Now we compare SDS cases listed in Table 4.2. Since $\frac{c_4}{s_2} \ll 1$, this leads to $\frac{c_4}{s_3} \ll 1$, and $\frac{c_4}{s_4} \ll 1$, and $\frac{c_4}{m} \ll 1$. Therefore $T_{SDS_4} > T_{SDS_3} > T_{SDS_2}$. Thus we drop cases SDS_3 or SDS_4.

  We drop case SDS_2 because $T_{SDS_2} > T_{DSD_2S_2}$. We also drop case SDS_1 because $T_{SDS_1} > T_{DSD_2S_1}$ given $\frac{\eta c_4}{d_2} \ll 1$.

- For DSDS, since $\frac{\eta c_4}{d_2} \ll 1$, $T_{DSD_2S_1} \approx 1 + \eta + \frac{c_2}{s_1}$. Then $T_{DSD_2S_1} < T_{DSD_3S_1}$ and $T_{DSD_2S_1} < T_{DSD_4S_1}$. Thus we drop cases DSD_2S_1 and DSD_4S_1.

  Since $\frac{c_4}{s_2} \ll 1$, $\frac{c_2}{s_2} \ll 1$. Then $T_{DSD_2S_2} \approx 1 + \eta + \eta c_2$. Then $T_{DSD_2S_2}$ is smaller than any of $T_{DSD_3S_2}$, $T_{DSD_4S_2}$, $T_{DSD_3S_3}$, $T_{DSD_4S_3}$, and $T_{DSD_4S_4}$. We drop all cases in DSDS except DSD_1S_1, DSD_2S_1, and DSD_2S_2.

  Since $T_{DSD_2S_2} > 1 + \eta + \eta\frac{c_2}{d_1} \approx T_{SDS_2D_1}$ given $\frac{\eta c_4}{s_2} \ll 1$, we can further drop case DSD_2S_2. Then $T_{DSD_1S_1} > T_{DSD_2S_1}$ and we can further drop case DSD_1S_1.

- For SDSD, we drop case SDS_1D_1 because $T_{SDS_1S_1} \approx 1 + \eta + \frac{c_4}{s_1} > T_{DSD_2S_1}$ given $\frac{\eta c_4}{d_2} \ll 1$. We drop Case SDS_3D_1 because $T_{SDS_3S_1} \approx 1 + \eta + \eta \cdot \frac{c_3}{d_1} > T_{SDS_2S_1}$. We drop Case SDS_4D_1 because $T_{SDS_4S_1} \approx 1 + \eta + \eta \cdot \frac{c_4}{d_1} > T_{SDS_2S_1}$.

  Note that $T_{SDS_2S_2} \approx 1 + \eta + \alpha_{2,2}c_2$ because $\frac{\eta c_4}{s_2} \ll 1$ and $\frac{\eta c_2}{d_2} < \frac{\eta c_4}{d_2} \ll 1$. Then we drop SDS_4D_4 because $T_{SDS_2D_2} < T_{SDS_4D_4}$. Also, $T_{SDS_2D_2}$ is smaller than any of $T_{SDS_3D_2}$, $T_{SDS_4D_2}$, $T_{SDS_3D_3}$, and $T_{SDS_4D_3}$. That is because $\alpha_{2,2} <$

$\alpha_{3,2}, \alpha_{4,2}, \alpha_{3,3}, \alpha_{4,3}$ due to the fact that s scorers fits in the smaller cache in Case SDS_2D_2 than other cases compared here. Thus only SDS_2D_2 and SDS_2D_1 qualify for the best candidates.

4.5 Evaluations

4.5.1 Settings

This section provides an experimental comparison of different cache blocking methods and validates the effectiveness of the selected method with unoptimized ones. The evaluation tasks are listed as follows: (1) Illustrate the fast comparison of the 28 range cases for using DSD, SDS, SDSD and DSDS with guided sampling. (2) Integrate our cache blocking selection algorithm with the QS algorithm [76] for tree-based ranking. (3) Assess the batched query processing in improving the throughput when n is small.

We implement the blocking methods using C compiled with GCC optimization flag -O3. Experiments are conducted on a Linux CentOS 6.6 server with 8 cores of 3.1GHz AMD Bulldozer FX8120 and 16GB memory. FX8120 has 16KB of L1. We set $L2 \approx 1MB$ as 2MB L2 cache is shared by two cores. Its 8MB L3 cache is shared by 8 cores and since L3 hosts tree data useful for multiple queries, we set $L3 \approx 2MB$. The cache line is of size 64 bytes. For AMD Bulldozer, c_2 is

around 7.3, c_3 is around 25.1, and c_4 is around 80.9. We have also conducted experiments in a 24-core Intel Xeon E5-2680v3 2.5 GHz server with $L1 = 32KB$, $L2 = 256KB$,, and $L3 \approx 2.5MB$ per core. The Intel results are similar and thus we mainly report the AMD numbers.

The following learning-to-rank datasets are used as evaluation benchmarks. (1) Yahoo! dataset [33] with 700 features per document feature vector. (2) MSLR-30K dataset [2] with 136 features per document vector. (3) MQ2007 dataset [1] with 46 features per document vector. Table 4.5 shows the range of d values when fitting d vectors in different cache levels for these 3 datasets.

d vectors fit in	Yahoo!	MS	MQ
L1	$d \leq 5$	$d \leq 30$	$d \leq 89$
L2	$d \leq 373$	$d \leq 1928$	$d \leq 5720$
L3	$d \leq 747$	$d \leq 3856$	$d \leq 11440$
Memory	$d \leq n$	$d \leq n$	$d \leq n$

Table 4.5: The vector counts for fitting in differnt cache levels.

We use LambdaMART [24] for ranking with additive tree ensembles and derive tree ensembles using the open-source jforests [51] package. To assess score computation in presence of a large number of trees, we have also used a bagging method [87] to combine multiple ensembles and each ensemble contains additive

boosting trees. Because the size of a scorer affects the cache performance and parameter choices, we generate the size of each tree with several settings: 10 leaves per tree, 50 leaves per tree, and 150 leaves per tree. Table 4.6 shows the range of s values when fitting s scorers in different cache levels under three choices of the regression tree size. The η value is about 1 because the basic access operation of a scorer is to fetch 1 tree node and then a document feature. Each of them fits in one cache line. The default total number of trees used is about 20,000 for Yahoo! dataset, 10,000 for MS, and 4,000 for MQ. We also use other numbers of trees in our experiments. When using the QS method [76], each scorer is a meta tree merged from multiple trees and η value is around 4 because the basic access operation of a scorer fetches elements from 4 data structures and then a document feature.

s scorers fit in	10 leaves	50 leaves	150 leaves
L1	$s \leq 25$	$s \leq 5$	$s = 1$
L2	$s \leq 1638$	$s \leq 327$	$s \leq 109$
L3	$s \leq 3276$	$s \leq 655$	$s \leq 218$
Memory	$s \leq m$	$s \leq m$	$s \leq m$

Table 4.6: The tree counts for fitting different cache levels.

The above data sets contain 23 to 120 documents per query with labeled relevancy judgment. In practice, a search system with a large dataset ranks thousands or tens of thousands of top results after a preliminary selection. To evaluate the score computation in such a setting, we synthetically generate more matched document vectors for each query. In this process, we generate relatively more vectors that bear similarity to those with low labeled relevance scores, because a large percentage of matched results per query are less relevant in practice. The number of vectors per query including synthetically generated vectors varies from 3,000 to 10,000 for Yahoo! dataset, from 2,000 to 6,000 for MS, and from 1,000 to 4,000 for MQ.

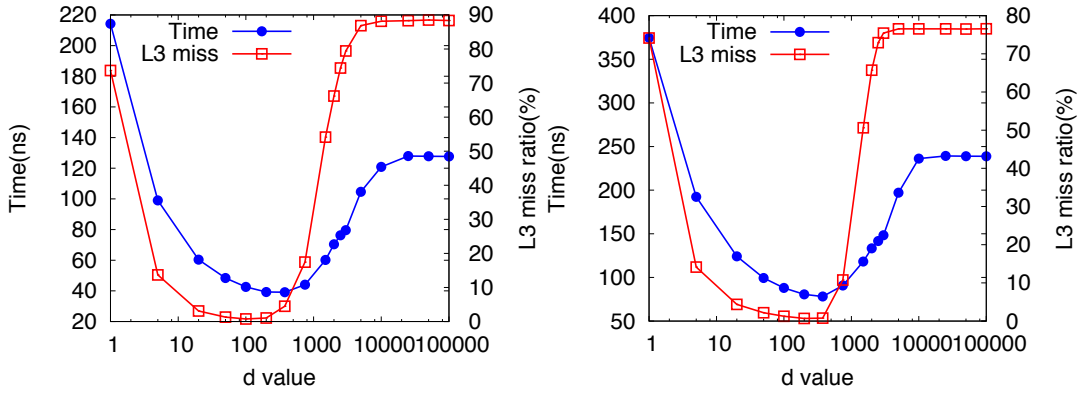
Metrics. We mainly report the average time of computing a subscore for each vector under one tree. With n matched vectors scored using an m-tree model, this scoring time multiplied by n and m is the scoring time per query. The throughput is the number of feature vectors scored per second. The number reported here is measured in a multi-core environment where each query is executed in a single core.

4.5.2 A Comparison of Cache Blocking Methods

Table 4.7 shows the score computing time of a vector per tree in nanoseconds under different cache blocking cases for Yahoo!, MS and MQ datasets. "Y! 10"

means Yahoo! dataset and each regression tree has 10 leaves. Row 2 is the scoring time of DS without cache blocking. The cost of all 28 cases under 4 cache blocking methods using guided sampling are listed, starting from Row 3. Under Proposition 1, our scheme searches the optimum only from four cases DSD_2, DSD_2S_1, SDS_2D_1, and SDS_2D_2. The corresponding entries in this table are marked in a gray color. For Yahoo! dataset with 150 leaves per tree, as we discussed in Section 4.4.5, extra two cases SDS_3D_1, and SDS_3D_2 are also compared and thus marked in a gray color. For each column from column 2, entry marked '$\star$' indicates the smallest value is found and this entry is considered to be highly competitive. Our comparison scheme selects DSD_2 as the best range case with $d = 373$ for Yahoo! dataset under all three tree size settings, $d = 1928$ for MS 50 leaves case and $d = 5720$ for MQ 10 leaves case. It selects SDS_2D_2 with $d = 1928$ and $s = 1638$ for MS 10 leaves case.

The running cost of the above guided sampling in CPU hours with one core is shown the second row of Table 4.8 and can be completed within about 10 hours using a 8-core server. We have also conducted exhaustive search with greatly-increased sampling points to obtain an estimated optimum solution. The best cases identified in the estimated optimum are listed in the third row of Table 4.8 and exactly match what has been selected by our guided sampling scheme. The fourth row of the Table 4.8 shows the sample error which is the cost difference ra-



(a) Yahoo! 50 leaves

(b) Yahoo! 150 leaves

Figure 4.5: Time cost and cache miss of DSD as d varies.

tio between the optimum solution and the solution approximated by our scheme. The difference is within 2.2%. The fifth and sixth rows are the best cases and difference ratio obtained on the Intel machine. The error is within 2.4% while all best cases of the estimated optimum match those of the approximated solution. The above result shows that our guided sampling can find a highly competitive blocking solution within reasonable hours using a modest server and such a solution can result in upto 6.57x response time reduction compared to DS without cache blocking.

Impact of blocking size on time cost and cache miss rate. In Section 4.4, we have used Figure 4.3 to illustrate the correlation between data access time cost and blocking size in deriving the cost for DSD. Figure 4.5 shows the experimental result to validate. This figure shows time cost curve of DSD and L3

	Y! 10	Y! 50	Y! 150	MS 10	MS 50	MQ 10
DS	59.83	214.29	374.56	59.67	206.98	34.89
DSD_1	33.04	99.00	193.95	17.34	47.79	9.82
DSD_2	14.09⋆	39.11 ⋆	78.11⋆	14.37	31.49⋆	8.52⋆
DSD_3	25.51	71.45	143.55	25.95	49.72	13.84
DSD_4	54.06	126.32	241.47	42.37	77.75	20.13
DSD_1S_1	41.56	125.44	237.35	24.46	63	13.67
DSD_2S_1	21.55	50.41	84.87	17.31	38.61	11.4
DSD_3S_1	25.71	77.75	141.07	18.1	40.38	11.6
DSD_4S_1	40.13	120.84	235.36	19.93	52.93	11.83
DSD_2S_2	29.77	72.42	123.3	30.38	67.33	19.5
DSD_3S_2	30.49	72.79	124.16	31.25	67.71	19.88
DSD_4S_2	29.73	72.74	123.07	31.35	68.45	19.85
DSD_3S_3	39.58	79.73	131.6	40.73	76.2	22.39
DSD_4S_3	46.13	105.91	157.47	46.2	100.58	28.16
DSD_4S_4	59.81	217.14	375.24	59.09	210.08	34.95
SDS_1D_1	60.67	149.17	267.69	23.67	62.61	10.99
SDS_2D_1	23.75	58.28	102.93	16.28	40.53	9.52
SDS_3D_1	27.08	72.31	130.48	16.85	43.22	9.6
SDS_4D_1	31.58	98.72	192.1	17.47	48.65	9.83
SDS_2D_2	14.5	39.75	82.26	13.21⋆	32.23	8.68
SDS_3D_2	15.71	42.68	88.05	15.09	37.30	10.12
SDS_4D_2	15.55	45.46	88.45	15.07	34.52	10.03
SDS_3D_3	20.76	59.03	120.15	21.55	50.02	13.04
SDS_4D_3	26.28	73.81	143.4	26.28	52.29	13.93
SDS_4D_4	54	131.58	250.17	42.46	81.7	20.13
SDS_1	42.96	113.73	240.89	21.39	50.83	12.62
SDS_2	30.91	72.22	122.22	31.65	67.28	20.27
SDS_3	46.11	107.19	156.67	46.56	100.75	28.78
SDS_4	59.83	214.29	374.56	59.67	206.98	34.89

Table 4.7: Scoring time of one vector per tree in nanoseconds for different cache blocking range cases.

miss rate measured using Linux tool perf when d value varies for Yahoo! dataset with 50 leaves (a) and 150 leaves (b) per tree. When d is too small, cache is not fully utilized and the cost of T_S for DSD derived in Section 4.4.1.2 is large. When d is too big which falls into case DSD_3 or DSD_4, the cost curve matches the analysis that $T_{DSD_4} > T_{DSD_3} > T_{DSD_2}$.

There is also a correlation between the overall time cost and L3 cache miss when d varies. For small d, the coefficient for L3 cost c_3 in T_S is big. For large d, there is more L3 cache miss, making T_D bigger as shown in Section 4.4.

Impact of m and n values on time cost. Figure 4.6 shows the time cost per tree per document when m changes from 2,000 to 20,000 for all datasets. In this experiment, we generate extra trees for MS and MQ datasets. It shows that with sufficiently large value of m, the cache behavior does not change much and the processing cost is about the same for different m values.

Figure 4.7 shows the time cost per tree per vector when n changes from 1 to 100,000. When n is smaller than 100, the performance drops significantly and cache is not fully utilized. When n is larger than 1000, the cost becomes stable and there is no much reduction. We will discuss the experiment results when batched query processing is allowed shortly.

	Y! 10	Y! 50	Y! 150	MS 10	MS 50	MQ 10
Comp. time	10.67h	27.09h	81.86h	2.719h	6.349h	0.424h
Best AMD	DSD_2	DSD_2	DSD_2	SDS_2D_2	DSD_2	DSD_2
Error AMD	0.21%	0.15%	0.10%	0.61%	2.2%	0.47%
Best Intel	DSD_2	SDS_2D_2	SDS_2D_2	DSD_2	SDS_2D_2	DSD_2
Error Intel	1.4%	2.4%	0.64%	0.18%	0.52%	0.14%

Table 4.8: CPU hours for comparison, sampling errors, and best cases.

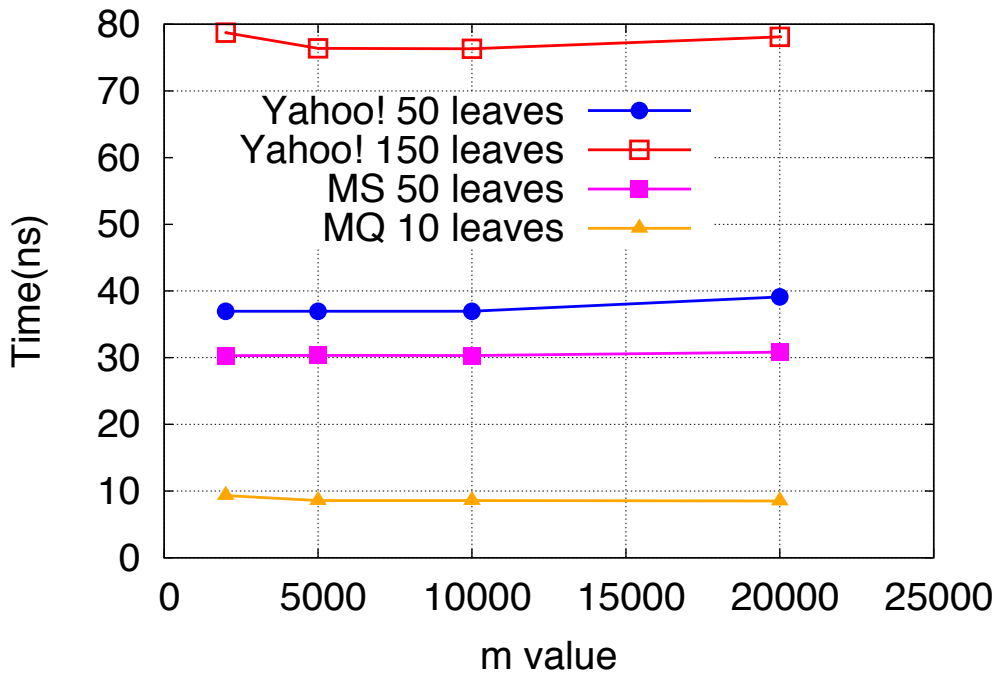


Figure 4.6: Scoring time per vector per tree when m changes.

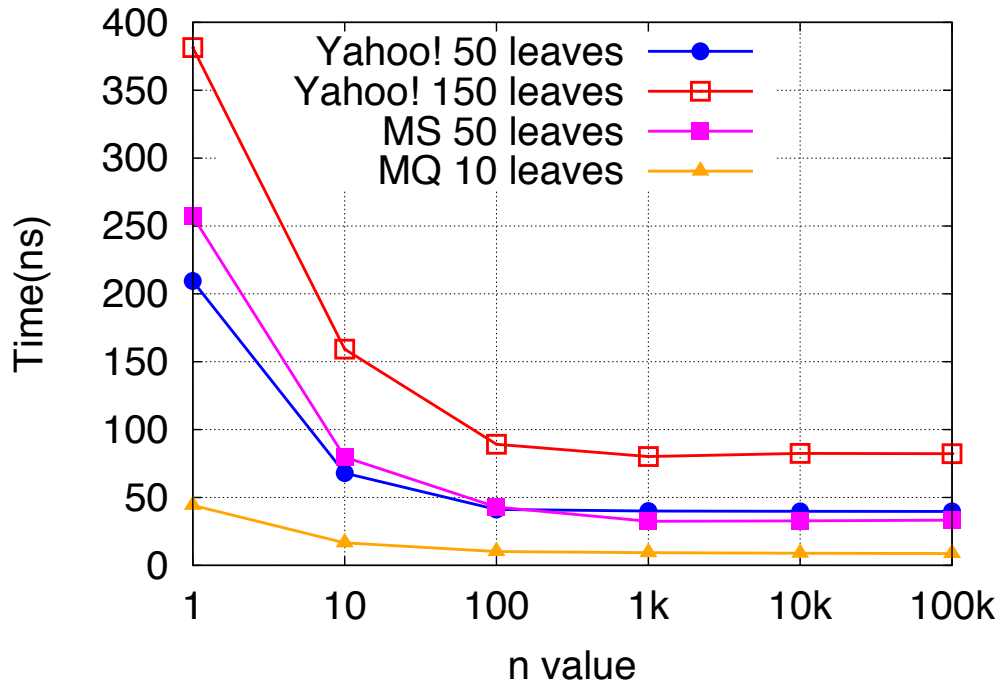


Figure 4.7: Scoring time per vector per tree when n changes.

Time(ns)	Tree scorers	BWQS scorers
Yahoo! 10	14.09	11.24
Yahoo! 50	39.11	52.88
Yahoo! 150	78.11	2869.29
MS 50	31.49	44.64
MQ 10	8.52	7.39

Table 4.9: Use of the comparison and selection scheme with BWQS scorers and with the original regression tree scorers.

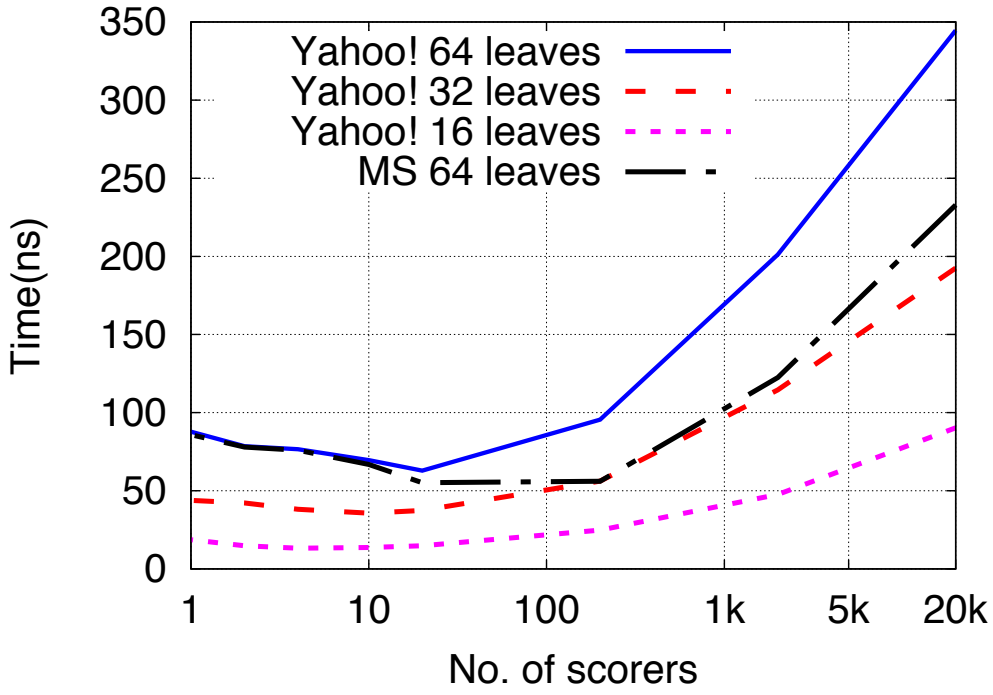


Figure 4.8: Scoring time of a vector per tree when varying the number of BWQS scorers.

4.5.3 Selective Cache Blocking for QuickScorer

We integrate our scheme with the BWQS algorithm [76] as follows. Step 1: Given m trees and let τ be the number of trees that will be merged to use the QS method. The number of scorers is $m' = m/\tau$. Step 2: Given m' scorers and n vectors, use our scheme to find the best blocking method and parameters. Step 3: Repeat Step 1 and Step 2 for a different sampling choice of τ. Step 4: The τ with the smallest time cost yields the best overall performance.

Figure 4.8 shows the BWQS results under the different number of scorers m' where m is fixed as 20,000 and m' varies from 1 to 20,000. Notice that when m' is small, each scorer contains many trees and does not fit in L1 or even L2. The best m' found for Y!64, Y!32, Y!16 and MS64 are 20, 10, 4 and 200, respectively. For Y!64, when $m' = 20$, case DSD_3 with $d = 75$ reaches the best performance with 62.89ns scoring time. If d is chosen as 1, the scoring time would be 89.33ns. Here the constraint to derive d value is explained as follow. Let F be the number of features in each document vector and L be the number of leaves in each tree. Following [76], the size of d document vectors is $4dF$ bytes and the QS data structure is composed of 6 parts: the result bit vectors with size $d\tau \cdot \frac{L}{8}$, the thresholds with size $4\tau L$, the offsets with size $4F$, the tree ids with size $4\tau L$, the bitvectors with size $\frac{\tau \cdot L^2}{8}$, and leaves with size $4\tau L$. The total size of d vectors and QS data structure is $4F(d+1) + (d \cdot \frac{L}{8} + (12 + \frac{L}{8})L) \cdot \tau$, which needs to fit in

L3 cache because one BWQS scorer may not fit in $L2$. For Yahoo! dataset with $F = 700$, when $L = 64$ and $\tau = 1000$, we can derive $d = 75.4$ with $L3 \approx 2MB$.

Table 4.9 shows the scoring time per vector per tree when applying cache blocking with our comparison and selection scheme to the original tree scorer and to the BWQS scorer. This comparison shows that when the number of leaves per tree is small(10), BWQS performs better. When the number of leaves per tree increases to 150, BWQS becomes fairly slow. Our explanation is listed as follows. The core QS scheme has a complexity sensitive to the number of tree nodes detected as "false" nodes because bit-wise operations need to be conducted for all such nodes. When the number of "false" nodes is large and linear to L, the overall time cost grows at linearly to increasing of L for small L. When $L > 64$, the bit operation has to be carried by multiple 64-bit instructions and there is additional overhead for managing this complexity. For a large tree with many false nodes, QuickScorer can become very expensive. On the other other hand, the original regression tree algorithm has a complexity logarithmically proportional to L. It should be mentioned that the scoring time per vector per tree reported here seems to be slower than what was reported in [76] for $L = 64$. That can be caused by a difference in dataset characteristics, our code implementation, and test platform. We will investigate this issue in the future work.

4.5.4 Batched Query Processing

Batch size	Y! 50	Y! 150	MS 50	MQ 10
10k	125.79	60.78	310.27	2880.2
1k	125.60	60.64	304.88	2805.8
100	125.00	60.37	308.45	2673.8
10	121.54	56.03	232.94	2460.6
1	73.53	31.40	125.33	1505.1
DS	23.87	13.12	38.89	565.6

Table 4.10: Throughput under different batch size when $n = 10$.

We illustrate the benefit of batched query processing when n is small. Table 4.10 shows the throughput under different batch sizes when ranking only 10 document vectors ($n = 10$). The throughput is defined as the number of queries processed per second. The last row shows the throughput when DS without cache blocking is used. When batch size is 10, for MS 50, the average processing time is reduced from 79.79ns to 42.93ns. When the batch size becomes much bigger, the benefit is not significant any more while there is an increase of waiting time. Thus a modest batch size is sufficient in this case to reach upto 1.86x throughput performance improvement.

4.6 Summary

The main contribution of this work is a fast comparison and selection scheme to find an optimized cache blocking method with guided sampling. Our analysis estimates the data access cost of different methods approximately, which provides a foundation to select sampling points in comparing different methods and in narrowing search space.

The evaluation studies with 3 datasets show that different blocking methods and parameter values can exhibit different cache and cost behavior and our guided sampling can identify a highly competitive solution among DSD, SDS, DSDS, and SDSD methods in a reasonable amount of hours using a modest multi-core server. The difference between the selected solution and the estimated optimum is within 2.4% and the response time of this solution can be 6.57x faster than DS without cache blocking. The analytic cost analysis shows that the search space for datasets such as Yahoo!, MS, and MQ can be greatly narrowed by taking advantages of data and architectural characteristics. When the number of feature vectors per query is small, cache utilization is affected and if allowed, batched query processing can bring upto 1.86x performance improvement. The evaluation demonstrates that our scheme can be used to find the optimized partitioning for QuickScorer.

Chapter 5

Conclusion

In this dissertation, we have sought the opportunities for the developement of information retrieval especially multi-version search and cache-conscious optimization, and we have also faced the challenges from scalability, efficiency and accuracy. We have thoroughly developed a new multi-version search architecture with fast ranking mechanism, where in Chapter 2 we present the new multi-version search system, in Chapter 3 we talked about how to speed up the core module of the online part of a search system, ranking, by proposing a new 2D block-based cache-conscious algorithm, and in Chapter 4 we provide a new framework with full cache analysis to help find best algorithm and parameter setting for our 2D block-based algorithm. We believe that our architecture and algorithm is general and can be used in both research and industry.

In this chapter, we first summarize our work on multi-version search and cache-conscious ranking optimization. We then share our lessons and wisdom learned from this work. Hopefully, our work can provide useful guidence and insights for researchers working in this domain. Finally, we discuss future work to conclude the chapter.

5.1 Summary

Thanks to the fast developement of information retrieval especially the proliferation of versioned search, organizations and companies archive many versions of digital data and this increases the needs to not only back up but also archive versioned data. This brings in many opportunities and challenges in this domain. Different from traditional search, versioned data has lots of redundancies between versions. How to inherit the property of redundancy is the key to tackle this problem. In this dissertation, we look at the problem from a system perspective, where we propose a new multi-version search architecture with representative cluster-based two-phase hybrid index, we revisit the key component of online search, the ranking module, and develope a new 2D block-based algorithm which can speedup ensemble ranking tremendously, and we present a new framework with full cache analysis to help find best algorithm and parameter setting for

our 2D block-based algorithm. In all the issues we looked at, we develope novel solutions and evaludate their effectiveness with several real-world datasets.

Firstly, we look at the state-of-the-art multi-version search architecture, where a two-phase approach [56, 89] has been proposed to find top results first using a non-positional index and then rerank the selected top results with a positional index. We find the problem is that still there is a large number of versions to go through in Phase 1 even without a positional index, so we are considering proposing a new architecture. In Phase 1, we use representatives of document versions with full positional information to reduce the number of top clusters needed to retain a good relevancy. In Phase 2, we extend the concept of cluster-based retrieval [4, 73, 75, 106] for representative-guided two-phase search and develop a per-cluster hybrid index to localize data access. The tradeoff is that Phase 2 requires memory caching of index or the use of solid state drives (SSD). To speedup Phase 2 search, we develop hybrid per-cluster indexing with adaptive traversal of forward and inverted structure. Finally, we evaluate our architecture on three real-world datasets which shows the effectiveness of our method.

Next, we look at the key component of online search within the whole search architecture, the ranking module. We find that among the whole online process, ranking is usually a significant time-consuming component and we are aiming at optimizing it. Gradient Boosted Regression Tree is the state-of-the-art learning-

to-rank algorithm which achieves best tradeoff between accuracy and efficiency and is widely used in industry nowadays. However, computing scores from a large number of trees is time-consuming. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and instruction branch prediction. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel. We find that unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. Thus, we ask a question: how can fast multi-tree ensemble ranking with simple code structure be accomplied via memory hierarchy optimization, without compromising ranking accuracy? We focus on this and propose a cache-conscious 2D blocking method to optimize data traversal for better temporal cache locality. Our evaluation shows that 2D blocking can be much faster than the baseline methods.

Last, we tackle problems on selecting best algorithm and parameter settings for the 2D blocking method. 2D blocking shows better cache locality and is much more efficient than traditional methods. However, there are other blocking methods to select and it is an open problem how to identify the best cache blocking method and parameter settings given different data and architecture characteristics. Experimentally determining this choice can be extremely time-consuming

and the comparative result may not be valid any more with a change of underlying feature vector structure or architecture. Thus, we provide an analysis of multiple blocking methods with different data traversal orders, which provides better insights on program execution performance and leads a fast approximation to select the optimized structure. The main contribution is that we have developed an analytic framework to compare memory access performance of data traversal under multi-level caches to find the fastest program execution with effective use of memory hierarchy. Our experiments with three datasets corroborate the effectiveness of search cost reduction while the guided approximation identifies a highly competitive blocking choice. We also demonstrate the use of this scheme with QuickScorer [76] and for batched query processing.

5.2 Lessons

Through the study of multi-version search and cache-conscious ranking optimization, we have learned three lessons. In this section, we will summarize them, and hopefully it can provide guidence for researchers in this direction.

Balance Tradeoffs in System Design. Large systems often face the challenges from different aspects. For example, for a search engine, on one hand we need to maintain high accuracy because users care a lot on the quality of the final ranking results. On the other hand, efficiency is also a key aspect. Nowa-

days, the online search is usually done within hundreds of milisecond to provide great user experience. However, balancing different tradeoffs is usually difficult because they natually contradict with each other. Therefore, we need to balance them according to the requirement of different projects and arrange the priority accordingly.

For example, in Chapter 2 we design a multi-version system using this philosophy. Comparing the baseline one phase approach (OP), we can be up-to two order of magnitude faster than them, while our accuracy is lower than them when we achieve the best efficiency. Comparing another baseline two phase approach (TP), its accuracy is low when it achives high efficiency so our accuracy can beat them tremendously, while the efficiency does not beat them like OP's two order of magnitude, but we are still up to 4 times faster than TP. Thus, we propose a new system which is better than both OP and TP by balancing the importance of different aspects.

Our lesson learned is that when building a real world large system, it is usually hard to beat all previous systems from all perspectives. How to choose the key aspect to optimize is very critical. Using the idea of balancing tradeoffs benefits a lot for our system design.

Apply Ideas from Other Research Domains. Avoid "reinventing wheel" again. There are many great research work in other domains and we can always

seek help from them to borrow ideas for our own problem. It is critical to understand the concept, background, challenges of our problem and the difference between ours to others.

One example is in Chapter 3 that we need to optimize the ensemble ranking problem. After unveiling the issue, we find one major problem lies in the data visit pattern. More specifically, there are too many cache misses in the algorithm. Thus, to enhance cache locality, we borrow the idea from other domains like computer achitecture and database optimization, where many previous work focused on improving cache locality by modifying loop orders and data visit patterns.

However, we also notice there are several differences between our work and the previous work. Firstly, we are aiming at an online ranking problem, so the number of features and ensemble size can change frequently. This leads to a key problem on how to find the best parameters because the efficiency is greatly dependent on choosing the block size properly, but a brute-force way to choose the best parameter takes too long. Therefore, in Chapter 4 we propose a new framework with complete cache analsysis to guide the parameter settings for our 2D block-based algorithm. Secondly, since its a ranking problem, we not only care about the average speed of one document ranked on one tree, we also care about the throughput of our system. Thus, we extend our 2D block-based algorithm to handle this issue.

Our lesson learned is that we can apply ideas from other research domains, but we also need to understand our problems deeply and consider the difference between our problem and the other ones carefully.

New Computer hardware Provides New Opportunities. Computer hardwares update in a fast pace. Some old ideas which do not work well on traditional hardware might work very well on new hardware and bring in opportunities. Thus, new hardware is like a revolution and might bring some "out-dated" ideas/research work back to our sights.

For example, in Chapter 2 when we design a two-phase hybrid index, we benefit a lot using the traditional concept of cluster-based index. This idea was proposed decades ago but did not end up as the main stream of versioned search. One of the reasons is on traditional HDD disk, the disk random IO is very costly and the cluster-based index brings in many random IOs. Therefore, although this idea has many merits in it, it was abandoned by the main stream of versioned search academia. However, with the developement of disk hardwares especially the widely used SSD disk, the random disk IO overhead is not that significant, so we bring back the idea of cluster-based index and reuse it on SSD disks. This gives us a lot of advantage under new hardware era.

Our lesson learned is that new computer hardware brings in new opportunities. We might be able to reuse some "out-dated" ideas under new hardware environment.

5.3 Future Work

In this dissertation, we discussed our work on multi-version search and cache-conscious ranking optimization. Because of the fast development in information retrieval especially versioned search and online ranking, we need to meet the challenges from different perspectives. Here, we discuss three potential future work in this section: multi-version search extension, cache-conscious method on other models and secure versioned search.

5.3.1 Multi-Version Search Extension

Our proposed multi-version search system is focused on conjunctive queries. One future study is to consider disjunctive queries because in real scenario, disjunctive queries will include more search results and can potentially increase search accuracy especially for queries with less matching documents.

Another future study is to investigate the incremental index update with time-based partitioning. When a new version is added, fragments shared with other

document versions need to be identified and an approximation under a certain time interval may be applied for cost reduction. The Phase 1 index may be changed following the traditional index update techniques if the cluster representative changes and the update for a cluster index is fairly local.

Also, the method of choosing representatives can be improved. Currently our recommended method is SLO, which includes the non-positional information from all versions and the positional information from the longest version. One potential future change is to add more positional information to the super document. For example, we can add phrases or sentences which appear in other versions but not the representative version to the super document. In this way, the super document will contain more positional information, which hopefully can improve search accuracy in Phase 1.

5.3.2 Cache-Conscious Method on Other Models

Our 2D blocking technique is studies in the context of tree-based ranking ensembles and one of the potential future work is to extend it for other types of ensembles, like Neural Networks and so on, by iteratively selecting a fixed number of the base rank models that can fit in the fast cache. Note that since our 2D block-based algorithm is a generic one, it should work for any ensemble based methods.

As to our proposed framework to choose best parameters and blocking methods using full cache analysis, for other ensembles we might meet the challenges like the ensemble size is not independant with the data size as we showed in the QS model. In this way, the cache analysis may be more complex and need further mathematics optimization.

Another future study is to investigate the cache behaviour under multi-thread environment. As to our tree-based ensemble, each tree and data entry is relatively small so we didn't see any difference from single-thread environment. However, as to other models where the ensemble is much larger than one thread's L1/L2 cache size, this might incur complex situation which needs to be investigated further because different thread might grab others' shared cache usage.

5.3.3 Secure Versioned Search

As sensitive information is increasingly stored on the cloud, privacy concerns have been a critical factor for users to adopt cloud-based information services. A dilemma commonly considered is that a user may not trust the capability that a cloud maintains for data privacy, yet the user also wants to take full advantage of the much larger resources of the cloud to search her or his data. Recently, there are lots of research work on secure search building on traditional index [95, 38, 32, 65, 70, 69, 30, 29, 31]. How to combine them with our new multi-version search

architecture to propose a new secure multi-version search system is an interesting problem.

Recall our system search pipeline, Phase 1 is just the same as traditional index so all the related work of secure search on traditional index should be able to migrated directly. Phase 2 uses cluster-based index. In each local cluster, we have fragment-version inverted index, forward index, reuse table and other auxiliary data structure. How to encrypt these needs further investigation. Also, currently, each cluster contains all versions of only one document. Since our system will go into k clusters in Phase 2, it knows which documents are highly relevant to the query and this is a severe leakage for secure multi-version search. One potential solution is to combine multiple documents' all versions to one cluster, say one cluster contains 100 or 1000 documents. Although this reduces the efficiency somehow, the server can now only guess the returned results with the probability of 1/100 or 1/1000. The higher the number of documents are in each cluster, the lower the probability that the server can guess correctly.

Bibliography

[1] Lector 4.0 datasets. http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx.

[2] Microsoft learning to rank datasets. http://research.microsoft.com/en-us/projects/mslr/.

[3] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning user interaction models for predicting web search result preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 3–10, New York, NY, USA, 2006. ACM.

[4] Ismail Sengor Altingovde, Engin Demir, Fazli Can, and Özgür Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Trans. Inf. Syst.*, 26(3):15:1–15:36, 2008.

[5] Avishek Anand, Srikanta Bedathur, Klaus Berberich, Ralf Schenkel, Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Efficient temporal keyword queries over versioned text. In *In Proc. of ACM CIKM Conf*, 2010.

[6] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *Proc. of 15th Australasian Database Conference*, pages 61–67, 2004.

[7] Peter G. Anick and Rex A. Flynn. Versioning a full-text information retrieval system. In *SIGIR*, pages 98–111, 1992.

[8] Avi Arampatzis and Jaap Kamps. A study of query length. In *Prc. of ACM SIGIR*, pages 811–812, 2008.

[9] Diego Arroyuelo, Senén González, Mauricio Marin, Mauricio Oyarzún, and Torsten Suel. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In *Proc. of 35th ACM SIGIR*, pages 255–264, 2012.

[10] Nima Asadi and Jimmy Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proc. of 36th ACM SIGIR*, pages 997–1000, 2013.

[11] Nima Asadi and Jimmy Lin. Training Efficient Tree-Based Models for Document Ranking. In *ECIR*, pages 146–157, 2013.

[12] Nima Asadi, Jimmy Lin, and Arjen P De Vries. Runtime Optimizations for Tree-Based Machine Learning Models. *IEEE TKDE*, pages 1–13, 2013.

[13] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[14] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. SIGIR '07, pages 183–190, 2007.

[15] Ricardo A. Baeza-Yates and Simon Jonassen. Modeling static caching in web search engines. In *ECIR*, pages 436–446, 2012.

[16] Ricardo A. Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[17] Jing Bai, Yi Chang, Hang Cui, Zhaohui Zheng, Gordon Sun, and Xin Li. Investigation of partial query proximity in web search. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1183–1184, 2008.

[18] Jing Bai, Jan Pedersen, and Mao Yang. Web-scale semantic ranking. In *Proceedings of the 2014 SIRIP*, 2014.

[19] Klaus Berberich, Srikanta Bedathur, Thomas Neumann, and Gerhard Weikum. A time machine for text search. In *Proc. of 30th ACM SIGIR*, pages 519–526, 2007.

[20] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[21] Leo Breiman and E. Schapire. Random forests. In *Machine Learning*, pages 5–32, 2001.

[22] Andrei Z. Broder, Nadav Eiron, Marcus Fontoura, Michael Herscovici, Ronny Lempel, John McPherson, Runping Qi, and Eugene J. Shekita. Indexing shared content in information retrieval systems. In *EDBT*, volume 3896, pages 313–330, 2006.

[23] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. ICML '05, pages 89–96, 2005.

[24] Christopher J. C. Burges, Krysta Marie Svore, Paul N. Bennett, Andrzej Pastusiak, and Qiang Wu. Learning to rank using an ensemble of lambda-gradient models. In *J. of Machine Learning Research*, pages 25–35, 2011.

[25] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proc. of 29th ACM SIGIR*, pages 621–622, 2006.

[26] B Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, and Jiang Chen. Early Exit Optimizations for Additive Machine Learned Ranking Systems Ranking in Additive Ensembles. In *WSDM*, pages 411–420, 2010.

[27] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. WSDM '10, pages 411–420, 2010.

[28] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. ICML '07, pages 129–136, 2007.

[29] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

[30] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric

encryption with support for boolean queries. In *CRYPTO 2013*, pages 353–373, 2013.

[31] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT 2014*, pages 351–368, 2014.

[32] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. ACNS'05, pages 442–455, 2005.

[33] Olivier Chapelle and Yi Chang. Yahoo! Learning to Rank Challenge Overview. *J. of Machine Learning Research*, pages 1–24, 2011.

[34] Francisco Claude, Antonio Fariña, Miguel A. Martinez-Prieto, and Gonzalo Navarro. Indexes for highly repetitive document collections. In *Proc. of 20th ACM CIKM*, pages 463–468, 2011.

[35] Francisco Claude and J. Ian Munro. Document listing on versioned documents. *LNCS*, 8214:72–83, 2013.

[36] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2010.

[37] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1:1–1:25, December 2010.

[38] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. CCS '06, pages 79–88. ACM, 2006.

[39] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proc. of VLDB*, 4(4):255–266, January 2011.

[40] Laura DuBois and Marshall Amaldas. Building the Case for Moving Compliance, eDiscovery, and Archives to the Cloud., June 2011.

[41] EMC. Data domain: Protection storage for backup and archive. http://www.emc.com/backup-and-recovery/data-domain/index.htm.

[42] EMC. Archive solutions for the enterprise with emc isilon scale-out nas. http://www.emc.com/collateral/white-papers/h11224-archive-solutions-enterprise-emc-isilon-wp.pdf, December, 2012.

[43] Kave Eshghi and Hsiu K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Hewlett-Packard Labs. Technical Report, TR 2005-30, 2005.

[44] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching

query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24:51–78, 2006.

[45] Héctor Ferrada and Gonzalo Navarro. A Lempel-Ziv compressed structure for document listing. *LNCS*, 8214:116–128, 2013.

[46] A. S. Fraenkel, S. T. Klein, Y. Choueka, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. of 9th ACM SIGIR*, pages 88–96. ACM, 1986.

[47] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, December 2003.

[48] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[49] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[50] Travis Gagie, Kalle Karhu, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Document listing on repetitive collections. *LNCS*, 7922:107–119, 2013.

[51] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. In *SIGIR*, pages 85–94, 2011.

[52] Pierre Geurts and Gilles Louppe. Learning to rank with extremely randomized trees. *J. of Machine Learning Research*, 14:49–61, 2011.

[53] Andrey Gulin, Igor Kuralenok, and Dmitry Pavlov. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. *J. of Machine Learning Research*, 14:63–76, 2011.

[54] Jinru He and Torsten Suel. Faster temporal range queries over versioned text. In *Proc. of 34th ACM SIGIR*, pages 565–574. ACM, 2011.

[55] Jinru He and Torsten Suel. Faster temporal range queries over versioned text. In *Proc. of 34th ACM SIGIR*, pages 565–574, 2011.

[56] Jinru He and Torsten Suel. Optimizing positional index structures for versioned document collections. In *Proc. of ACM SIGIR*, pages 245–254, 2012.

[57] Jinru He, Hao Yan, and Torsten Suel. Compact full-text indexing of versioned document collections. In *Proc. of 18th ACM CIKM*, pages 415–424, 2009.

[58] Jinru He, Junyuan Zeng, and Torsten Suel. Improved index compression techniques for versioned document collections. In *Proc. of 19th ACM CIKM*, pages 1239–1248, 2010.

[59] Sándor Héman. Super-scalar database compression between ram and cpu-cache. In *MS Thesis, Centrum voor Wiskunde en Informatica*, 2005.

[60] Michael Herscovici, Ronny Lempel, and Sivan Yogev. Efficient indexing of versioned document sequences. In *ECIR*, pages 76–87, 2007.

[61] HP. Backup, Recovery and Archive – Eliminate the boundaries of traditional data protection and retention. http://www8.hp.com/us/en/products/data-storage/storage-backup-archive.html.

[62] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: An empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, 1998.

[63] Symantec Inc. Your Backup Is Not an Archive , 2011.

[64] Internet Archive. http://www.archive.org.

[65] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

[66] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, October 2002.

[67] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 133–142, New York, NY, USA, 2002. ACM.

[68] K. Sparck Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, November 2000.

[69] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, pages 258–274, 2013.

[70] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS'12*, pages 965–976, 2012.

[71] Marcin Kaszkiel, Justin Zobel, and Ron Sacks-davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17:406–439, 1999.

[72] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In *ATEC '04: Pro-*

ceedings of the annual conference on USENIX Annual Technical Conference, pages 59–72, Berkeley, CA, USA, 2004. USENIX Association.

[73] Oren Kurland and Eyal Krikon. The opposite of smoothing: A language model approach to ranking query-specific document clusters. *J. Artif. Int. Res.*, 41(2):367–395, May 2011.

[74] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 102–111. ACM, 2003.

[75] Xiaoyong Liu and W. Bruce Croft. Cluster-based retrieval using language models. In *Proc. of 27th ACM SIGIR*, pages 186–193, 2004.

[76] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *SIGIR*, pages 73–82, 2015.

[77] Evangelos P. Markatos. On caching search engine query results. *Computer Communications*, 24:137–143, 2001.

[78] Donald Metzler and W. Bruce Croft. Linear feature-based models for information retrieval. *Inf. Retr.*, 10(3):257–274, June 2007.

[79] Donald Metzler and W. Bruce Croft. A markov random field model for term dependencies. In *Proc. of 28th ACM SIGIR*, pages 472–479. ACM, 2005.

[80] Teng-Sheng Moh and BingChun Chang. A running time improvement for the two thresholds two divisors algorithm. In *Proc of 48th ACM Ann. Southeast Regional Conf.*, pages 69:1–69:6, 2010.

[81] Athicha Muthitacharoen, Benjie Chen, and David Mazires. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001.

[82] NetApp. Data Archieve Solutions. http://www.netapp.com/us/solutions/data-protection/archive-solutions.aspx.

[83] Douglas W. Oard, Jason R. Baron, Bruce Hedin, David D. Lewis, and Stephen Tomlinson. Evaluation of information retrieval for e-discovery. *Artif. Intell. Law*, 18(4):347–386, December 2010.

[84] Rifat Ozcan, Ismail Sengr Altingvde, Berkant Barla Cambazoglu, Flavio Paiva Junqueira, and zgr Ulusoy. A five-level static cache architecture for web search engines. *Inf. Process. Manage.*, 48:828–840, 2012.

[85] Rifat Ozcan, Ismail Sengr Altingvde, and zgr Ulusoy. Static query result caching revisited. In *WWW*, pages 1169–1170, 2008.

[86] Rifat Ozcan, Ismail Sengr Altingvde, and zgr Ulusoy. Cost-aware strategies for query result caching in web search engines. *TWEB*, 5:9, 2011.

[87] Dmitry Yurievich Pavlov, Alexey Gorodilov, and Cliff A. Brunk. Bagboo: a scalable hybrid bagging-the-boosting model. In *CIKM*, pages 1897–1900, 2010.

[88] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Inf. Retr.*, 13(4):346–374, August 2010.

[89] Yves Rasolofo and Jacques Savoy. Term proximity scoring for keyword-based retrieval systems. In *In Proc. of the 25th European Conf. on IR Research*, pages 207–218, 2003.

[90] Transparency Market Research. eDiscovery (Software and Service) Market - Global Scenario, Trends, Industry Analysis, Size, Share and Forecast, 2010 - 2017. http://www.transparencymarketresearch.com/ediscovery-market.html, 2013.

[91] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD*, pages 76–85, 2003.

[92] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of 25th ACM SIGIR*, pages 222–229, 2002.

[93] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of SIGIR'02*, pages 222–229. ACM, 2002.

[94] IBM Global Technology Services. Cloud-based data archiving service. IBM White paper. July 2011 .

[95] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. SP '00. IEEE Computer Society, 2000.

[96] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000.

[97] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization. In *In Khalid Sayood, editor, Lossless Compression Handbook*. Academic Press, 2002.

[98] Krysta M Svore, Pallika H Kanani, and Nazan Khan. How good is a span of terms?: exploiting proximity to improve web retrieval. In *Proc. of 33rd ACM SIGIR*, pages 154–161, 2010.

[99] Xun Tang, Xin Jin, and Tao Yang. Cache-conscious runtime optimization for ranking ensembles. SIGIR '14, pages 1123–1126, 2014.

[100] Tao Tao and ChengXiang Zhai. An exploration of proximity measures in information retrieval. In *Proc. of 30th ACM SIGIR*, pages 295–302, 2007.

[101] Dan Teodosiu, Nikolaj Bjorner, Yuri Gurevich, Mark Manasse, and Joe Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. Microsoft Research TR-2006-157, 2006.

[102] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *CIKM*, pages 1209–1212, 2013.

[103] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *CIKM*, pages 1209–1212, 2013.

[104] Leong Hou U, Nikos Mamoulis, Klaus Berberich, and Srikanta Bedathur. Durable top-k search in document archives. In *Proc. of 2010 ACM SIGMOD*, pages 555–566, 2010.

[105] Leong Hou U, Nikos Mamoulis, Klaus Berberich, and Srikanta Bedathur. Durable top-k search in document archives. In *Proc. of ACM SIGMOD*, pages 555–566, 2010.

[106] Ellen M. Voorhees. The cluster hypothesis revisited. In *Proc. of 8th ACM SIGIR*, pages 188–196, 1985.

[107] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The impact of solid state drive on search engine cache management. SIGIR '13, pages 693–702.

[108] Lidan Wang, Jimmy Lin, and Donald Metzler. Learning to efficiently rank. SIGIR '10, pages 138–145, 2010.

[109] Yuedui Wang, Xiangming Wen, Yong Sun, Zhenmin Zhao, and Tianpu Yang. The Content Delivery Network System Based on Cloud Storage. *2011*

International Conference on Network Computing and Information Security, pages 98–102, May 2011.

[110] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, second edition*. Morgan Kaufmann Publishing, May 1999.

[111] Qiang Wu, Christopher J. Burges, Krysta M. Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, June 2010.

[112] Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 391–398, 2007.

[113] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of 18th Inter. Conf. on World Wide Web*, pages 401–410, 2009.

[114] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of 18th inter. Conf. on World Wide web*, pages 401–410. ACM, 2009.

[115] Tao Yang and Apostolos Gerasoulis. Web search engines: Practice and experience. in Computer Science Handbook (Teofilo Gonzalez, Jorge Diaz-Herrera, Allen Tucker. Eds), Chapman and Hall/CRC Press, 2014.

[116] Yisong Yue, Thomas Finley, Filip Radlinski, and Thorsten Joachims. A support vector method for optimizing average precision. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 271–278, 2007.

[117] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. WWW '08, pages 387–396.

[118] Jiangong Zhang and Torsten Suel. Efficient search in large textual collections with redundancy. WWW '07, pages 411–420.

[119] Jiashu Zhao and Jimmy Xiangji Huang. An enhanced context-sensitive proximity model for probabilistic information retrieval. SIGIR '14, pages 1131–1134. ACM, 2014.

[120] Jiashu Zhao, Jimmy Xiangji Huang, and Ben He. Crter: Using cross terms to enhance probabilistic information retrieval. In *Proc. of 34th ACM SIGIR*, pages 155–164, 2011.

[121] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[122] M. Zukowski, S. Hman, N. Nes, P. A. Boncz, Marcin Zukowski, Sndor Hman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proc. of IEEE ICDE*, page 59, 2006.

[123] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proc. of ICDE*, pages 59–59. IEEE, 2006.